A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java

Benjamin M. Brosgol
Ada Core Technologies
79 Tobey Road
Belmont, MA 02478 USA
+1.617.489.4027 (Phone)
+1.617.489.4009 (FAX)
brosgol@gnat.com

1. ABSTRACT

Both Ada and Java support concurrent programming, but through quite different approaches. Ada has built-in tasking features with concurrency semantics, independent of the language's OOP model, whereas Java's thread support relies on OOP and is based on special execution properties of methods in several predefined classes. Ada achieves mutual exclusion through protected objects with encapsulated components; Java uses the classical "monitor" construct with "synchronized" methods/blocks. Ada models condition-based synchronization and communication through suspension objects, protected entries, and rendezvous; Java provides the low-level "wait" / "notification" methods. Both languages offer timing control; Ada additionally provides user-specifiable scheduling policies.

Compared to Java, Ada's concurrency model is less susceptible to deadlock and race conditions. Neither language is intrinsically superior in run-time performance. In some areas Ada's semantics entail less run-time overhead; in other areas Java may have the advantage.

Ada offers direct support for real-time programming through a combination of facilities in the core language and the Systems Programming and Real-Time Systems Annexes. Java lacks some critical functionality, and its semantics for thread scheduling are not completely defined. Depending on dynamic allocation and garbage collection, Java raises issues of time and space predictability. Work is in progress on making Java more applicable to real-time systems; at present Ada is the more appropriate language.

1.1 Keywords

Ada, Java, concurrency, threads, tasking, real-time, inheritance anomaly, Object-Oriented Programming

2. INTRODUCTION

Ada [1] and Java [2] [3] are unusual in providing direct language support for concurrency: the *task* in Ada and the *thread* in Java. Although they offer roughly equivalent functionality – the ability to define units of concurrent execution and to control mutual exclusion, synchronization, communication, and timing – the two languages have some major differences. This paper compares and contrasts the concurrency-related facilities in Ada and Java, focusing on their expressive power, support for sound software engineering, and performance.

A brief comparison of concurrency support in Ada and Java may be found in [4]; this paper extends those results. A comprehensive discussion of Ada tasking appears in [5]; references targeting Java concurrency include [6] and [7].

The term "concurrent" is used here to mean "potentially parallel." Issues related to multiprocessor environments or distributed systems are beyond the scope of this paper.

Section 3 summarizes the Java language and technology. Sections 4 through 6 contrast task/thread "lifetime" issues in Ada and Java: declaration, creation/startup, and termination. Sections 7 and 8 deal with the most important elements of the languages' concurrency support, namely mutual exclusion and synchronization/communication. Section 9 focuses on the languages' approaches to real-time

This paper is a minor update (April 2000) to a version presented at the Ada UK Conference in Bristol, UK in October 1998 while the author was with Aonix. It is a revision of "A Comparison of the Concurrency Features of Ada 95 and Java", *SIGAda '98 Proceedings*, ©1998 Association for Computing Machinery, Inc., and is reprinted with permission.

support. Section 10 compares how the languages cope with the "inheritance anomaly", an interaction between concurrency and Object-Oriented Programming. Section 11 contrasts Ada and Java with respect to the interactions between exception handling and the concurrency features. Section 12 touches on some additional issues, and Section 13 presents the conclusions. An extended example in both Ada and Java, the bounded buffer, appears in an Appendix.

3. JAVA SUMMARY

This section provides an overview of the Java technology and the basic language features. Among the references for further information on Java are [2], [3], and [8]. Additional points of comparison between Ada and Java may be found in [4] and [9].

3.1 Java Technology Elements

Sun [10] has described Java as a "simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language". This is an impressive string of buzzwords – [8, pp. 3ff] summarizes how Java complies with these goals – but it is useful to distinguish among three elements:

- The Java language
- The predefined Java class library
- The Java execution platform, also known as the *Java Virtual Machine* or simply *JVM*.

In brief, Java is an Object-Oriented language with features for objects/classes, encapsulation, inheritance, polymorphism, and dynamic binding. Its surface syntax has a strong C and C++ accent: for example, the names of the primitive types and the forms for the control structures are based heavily on these languages. However, the OO model is more closely related to so-called "pure" OO languages such as Smalltalk and Eiffel. Java directly supports single inheritance and also offers a partial form of multiple inheritance through a feature known as an "interface".

A key property of Java is that objects are manipulated indirectly, through implicit references to explicitly allocated storage. The JVM implementation performs automatic garbage collection, as a background thread.

One can use Java to write stand-alone programs, known as *applications*, in much the same way that one would use Ada, C++, or other languages. Additionally, and a major reason for the attention that Java is currently attracting, one can use Java to write *applets* – components that are referenced from HTML pages, possibly downloaded over the Internet, and executed by a browser or applet viewer on a client machine.

Supplementing the language features is a comprehensive set of predefined classes. Some support general-purpose programming: for example, there are classes that deal with string handling, I/O, and numerics. Others, such as the Abstract Windowing Toolkit, deal with Graphical User Interfaces. Still others support specialized areas including distributed component development, security, and database connectivity.

There is nothing intrinsic about the Java language that prevents a compiler from translating a source program into a native object module for the target environment, just like a compiler for a traditional language. However, it is more typical at present for a Java compiler to generate a so-called *class file* instead: a sequence of instructions, known as "bytecodes", that are executed by a Java Virtual Machine. This facility is especially important for downloading and running applets over the Internet, since the client machine running a Web browser might be different from the server machine from shich the applet is retrieved. Obviously security is an issue, which is addressed through several mechanisms, including:

- Java language semantics
  - Type safety
  - Absence of pointers (and hence absence of insecurities from pointer manipulation)
  - Prevention of accesses to uninitialized variables
  - Run-time checking on array indexing and many other operations
- Security-related classes
- The implementation of the JVM, which performs a load-time analysis of the class file to ensure that it has not been compromised
- The implementation of the browser or applet viewer, which checks that a downloaded applet does not invoke methods that access the client machine's file system
- Signed applets

3.2 General-Purpose Language Features

At one level Java can be regarded as a general-purpose programming language with traditional support for software development. Its features in this area include the following:

- simple control structures heavily resembling those found in C and C++
- a set of primitive data types for manipulating numeric, logical, and character data
- a facility for constructing dynamically-allocated linear indexable data structures (arrays)
- a facility for code modularization ("methods")
- limited block structure, allowing the declaration of variables, but not methods, local to a method
- exception handling

The primitive data types in Java are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. Java is distinctive in specifying the exact sizes and properties of these primitive types. For example, an `int` is a 2's complement 32-bit quantity, with "wrap around" when an operation overflows. The floating-point types are defined with IEEE-754 semantics [11], implying that results and operands may be signed zeroes, signed infinities, and NaN ("Not a Number"). Unlike Ada, Java does not allow range constraints to be specified for variables from numeric types.

Java's `char` type is 16-bit Unicode. Its source representation is likewise base on Unicode, although external files may be stored in 8-bit format with conversion on the fly during file loading. Like C and C++ but unlike Ada, Java's treatment of identifiers is case sensitive.

Java lacks a number of data structuring facilities found in traditional languages:

- enumeration types
- heterogeneous data structures (records / structs)
- conditional data structures (variant records / unions)

An enumeration type can be simulated by constants (static "final" variables), a record/struct can be modeled by a class, and a variant record/union can be modeled by an inheritance hierarchy.

Significantly, Java also lacks an explicit pointer facility, a restriction motivated by security concerns. As a consequence, it is not possible to obtain a pointer to a method, functionality that languages like Ada and C/C++ provide for "callbacks" in GUI programming. There is a workaround to simulate callbacks, namely a reference to an object of a class that implements the method, but this is not as direct as the mechanisms in Ada and C/C++.

3.3 Java Run-Time Execution Model

Java's execution model is based on a run-time stack (more generally, one stack per created thread). When a method is called, its parameters and local variables are stored on the (calling thread's) stack. For a parameter or variable of a primitive type, the stack contains the variable's value directly. In all other cases the stack contains a *reference* that is either null or else designates an allocated object of an appropriate type. Parameter passing is thus always "by value", with the value of the actual parameter copied to the formal parameter at the point of call. However, since objects are represented indirectly, the effect is to copy a reference and thus the formal and actual parameters refer to the same object.

As will be discussed in detail below, Java offers built-in support for multi-threaded programs, with user-definable threads that can communicate through objects whose methods are explicitly marked as `synchronized`. An object that contains `synchronized` methods has a "lock" that enforces mutually exclusive access, with calling threads suspended waiting for the lock as long as any `synchronized` method is being executed by some other thread. The Java thread model is based on its OOP features.

3.4 "Programming in the large"

Java offers a variety of support for developing large systems. It provides a full complement of Object-Oriented features, with precise control over the accessibility of member names (that is, control over encapsulation) as will be summarized below.

The unit of program composition is the class, but this raises the problem of "namespace pollution" as large numbers of classes are developed. Java solves this in two ways. First, classes may be nested, a facility introduced in the Java 1.1 release. Thus a class may declare other classes as members; since the inner classes are referenced using "dot" notation, their names do not conflict with any other classes. Second, Java provides a namespace management feature, the *package*, as a repository for classes. Despite the similarity of names, the term "package" has different meanings in Ada and Java. An Ada package is a syntactic and semantic construct, a compatible unit. In contrast, a Java package is an open-ended host-environment facility, generally a directory, that contains compiled class files. Java supplies an explicit construct, the `package` statement, which allows the programmer to identify the target package for the classes being compiled. If a source file does not supply an explicit `package` statement, then its classes go into an "unnamed" package, by default the same directory as the site of the `.java` file.

Java's predefined environment is structured as a collection of packages, including `java.lang`, `java.util`, and many others. If a Java program explicitly imports a class or package, then it can access that class, or any class in the given package, by the class's "simple name" without the package name as qualifier. The general-purpose package `java.lang` is implicitly imported by every Java program, and its classes (such as `String`) are therefore automatically accessible without the package name as prefix.

Like Ada, but unlike C and C++, Java does not supply a preprocessor. Higher level and better-structured mechanisms provide the needed functionality more reliably than preprocessor directives.

Java does not include a facility for generic units ("templates" as they would be known in C++). Some of this functionality can be approximated in Java through use of the "root" class `Object` defined in package `java.lang`, but with much less compile-time protection and control than with Ada generics.

3.5 Java Application Example

To introduce the basic language constructs, here is a simple Java application that displays its command-line arguments:

```
class DisplayArgs{
  static int numArgs;
  public static void main( String[] args ){
    numArgs = args.length;
    for (int j=0; j < numArgs; j++) {
      System.out.println( args[j] );
    } // end 'for'
  } // end 'main'
} //end 'DisplayArgs'
```

The *class* is the unit of compilation in Java, and is also a data type; here it is only the compilation unit property that is being exploited. A class declares a set of *members*: in this example the class `DisplayArgs` has two members, a *field* named `numArgs` of type `int`, and a *method* named `main`.

The field `numArgs` is declared `static`, implying that there is a single copy of the data item regardless of the number of instances of its enclosing class.

The method `main` takes one parameter, `args`, an array of `String` objects. It does not return a value (thus the keyword `void`) and is invokable independent of the number of instances of its enclosing class (thus the keyword

static). The public modifier implies that the method name may be referenced anywhere that its enclosing class is accessible.

There is special significance to a method named main that is declared public and static, and that takes a String array as parameter and returns void as its result. Such a method is known as a class's "main method." When a class is specified on the command line to the Java interpreter, its main method is invoked automatically when the class is loaded, and any arguments furnished on the command line are passed in the args array. For example if the user invokes the application as follows:

```
java DisplayArgs Brave new world
```

then args[0] is "Brave", args[1] is "new", and args[2] is "world". Similar to C and C++, the initial element in an array is at position 0. Since the number of elements in an array is given by the array's length field, the range of valid indexes for an array x goes from 0 to x.length-1. Unlike C and C++, an array is not synonymous with a pointer to its initial element, and in fact there is a run-time check to ensure that the value of an index expression is within range. If not, an exception is thrown.

The class String is defined in the java.lang package. A variable of type String is a reference to a String object, and a string literal such as "Brave" is actually a reference to a sequence of char values; recall that char is a 16-bit type reflecting the Unicode character set. There is no notion of a "nul" character terminating a String value; instead, there is a method length() that can be applied to a String variable to determine the number of char values that it contains. String variable assignment results in sharing, but there are no methods that can modify the contents of a String. To deal with "mutable" strings, the class StringBuffer may be used.

The Ada and Java String types are thus quite different. In Ada, a String is a sequence of 8-bit Character values, and declaring a String requires specifying the bounds either explicitly through an index constraint or implicitly through an initialization expression; further, in an array of String values each element must have the same bounds. In Java a String is a reference to an allocated sequence of 16-bit char values, and a length needs to be specified when the object is allocated, not when the reference is declared. Further, as with the args parameter to main, in an array of String values different elements may reference strings of different lengths.

The principal processing of the main method above occurs in the for loop. The initialization part declares and initializes a loop-local int variable j. The test expression j < numArgs is evaluated before each iteration; if the value is true then the statement comprising the loop body is executed, otherwise the loop terminates. The loop epilog j++, which increments the value of j, is executed after each iteration.

The statement that is executed at each iteration is the method invocation

```
System.out.println( args[j] );
```

System is a class defined in the package java.lang, and out is a static field in this class. This field is of class PrintStream, and println is an instance method for class PrintStream. Unlike a static method, an instance method applies to a particular instance of a class, a reference to which is passed as an implicit parameter. The println method takes a String parameter and returns void. Its effect is to send its parameter and a trailing end-of-line to the standard output stream, which is typically associated with the user's display.

The example also illustrates one of Java's comment conventions, namely "//" through the next end-of-line.

3.6 Object-Oriented Programming in Java

Java is a "pure" Object-Oriented Language in the style of Smalltalk and Eiffel. The class construct serves three purposes:

- A compilable module
- A template for the construction of data objects allocated on the heap
- A type for a polymorphic variable

A Java *class* has *members*; a member may be a *field*, a *method*, or another class, and is either *per-class* or *per-instance*. Defining a member with an explicit modifier static makes it per-class, otherwise the member is per-instance.

As implied by the terminology, if a member field is per-class then there is exactly one copy regardless of the number of instances of the class. In contrast, there is a different copy of a per-instance field in each instance of the class.

If a method m() is a static member of class K, then it is invoked with the syntax

```
K.m( parameters )
```

and its only parameters are the ones it explicitly declares.

If m() is per-instance, then it is invoked with the syntax

```
ref.m( parameters )
```

where ref is a reference to an instance of either class K or some class that inherits (directly or indirectly) from K. The method takes an implicit parameter named this which refers to an object whose class contains the method as a member. The invocation of a per-instance method is *dynamically bound*: the version of m() that is called is based on the class of the object that ref references.

Most non-trivial classes have one or more *constructors*: special parameterizable forms that allow "clients" of the class to create (allocate) objects with controlled initialization.

Visibility control (and thus encapsulation) is obtained through *access modifiers* included in the declaration of classes, members, and constructors.

If a top-level class is declared public, then it is accessible anywhere its package is accessible. Otherwise it is accessible only from classes in the same package.

If a member or constructor in a class is declared `public`, then it is accessible wherever its class is accessible. If declared `protected`, it is accessible only from classes in the same package or from subclasses. If declared `private`, it is accessible only from within the class itself. If there is no access modifier, then the member or constructor has "package" accessibility and is accessible from any class in the same package, but not from outside.

A class may be declared to *inherit from* ("extend") at most one superclass. If no superclass is specified explicitly, then the class implicitly extends the predefined class `Object` from package `java.lang`. When a superclass is extended, all public and protected instance methods are automatically inherited by the subclass; the subclass may override any of these and may also declare new members and constructors. It is common for an overriding implementation of a superclass method `m()` to need to invoke the overridden method; the special form `super.m()` achieves this effect and is bound statically.

A class may be specified as *abstract*, implying that no objects of that class may be constructed. An abstract class may have abstract methods (lacking implementations); non-abstract subclasses of an abstract class need to provide implementations for any abstract methods that they inherit.

A variable declared of a particular class is intrinsically *polymorphic*: it can designate objects from that class or from any of its direct or indirect subclasses. Objects are generated on the heap from invocations of constructors.

Complementing the class concept is the feature known as an *interface*. An interface may be viewed as a restricted form of class with no "implementation" aspects. Thus the only methods allowed in an interface are abstract methods, and the only variables allowed in an interface are so-called *final* variables, which are constants. A class is allowed to extend only one class but may implement an arbitrary number of interfaces; in this sense Java provides some support for multiple inheritance.

Here is an example of Java's main OOP concepts, adapted from [4]:

```
public class Point{
  protected static int numPts=0;
  protected int x, y;

  public Point(int x, int y){
    this.x=x;
    this.y=y;
    numPts++;
  }

  public static int getNumPts(){
    return numPts;
  }
  public void shift(int Δx, int Δy){
    x += Δx;  // Note Unicode character
    y += Δy;
   }
  public void put(){
    System.out.println("x = " + x );
    System.out.println("y = " + y );
  }
}
```

The constructor for `Point` initializes the instance fields `x` and `y` and increments the static field `numPts`. Since `numPts` is protected, a "client" of `Point` cannot access this field directly. Instead, and more safely, it can retrieve the value through a call on the public static method `getNumPts()`. `Point` also declares two instance methods, `shift()` and `put()`, the former coincidentally illustrating Java's support for Unicode characters such as 'Δ'.

```
public class ColoredPoint extends Point{
  protected int color;

  public ColoredPoint( int x, int y,
                       int color ){
    super(x, y);
    this.color = color;
  }
  protected void invertColor(){
    color = -color;
  }
  public void put(){
    super.put();
    System.out.println("color = " + color);
  }
}
```

The `ColoredPoint` class extends `Point` and supplies a new constructor (which invokes its superclass's constructor), a new instance method `invertColor()`, and an overriding version of the inherited `put()`. The instance method `shift()` from `Point` is inherited and is not overridden.

```
public class Example{
  public static void main( String[] args ){
    Point p;
    p = new ColoredPoint(1, 2, 10);
    p.shift( 5, 6 );
    p.invertColor();  // Illegal
    ((ColoredPoint)p).invertColor();
    p.put();
    System.out.println("Point count = " +
                    Point.getNumPts() );
  }
}
```

The declaration of `p` does not create any objects; it reserves space for a (polymorphic) reference. After the assignment statement, `p` references an object of class `ColoredPoint`. The invocation of `shift()` dynamically binds to the version inherited from `Point`. The invocation of `invertColor()` directly on `p` is illegal; `p` needs to be cast to `ColoredPoint` (with a run-time check) in order for the method invocation to be legal. The invocation of `put()` dynamically binds to the overridden version of `put()` declared in `ColoredPoint`. The invocation of `getNumPts()` is statically bound.

The output of the above program (after the illegal statement has been removed) is:

```
x = 6
y = 8
color = -10
Point count = 1
```

4. TASK/THREAD DECLARATION

An Ada *task* is a unit of modularization comprising a specification and a body, and it is also a data object. A template for such objects is a *task type*. Here is a package that declares a task type, along with an access-to-task-type for dynamic allocation of task objects.

```
package Outputter_Pkg is
   task type Outputter;
   type Outputter_Ref is access Outputter;
end Outputter_Pkg;
```

The algorithm performed by each object of the type is an infinite loop that displays a string:

```
with Ada.Text_IO; use Ada.Text_IO;
package body Outputter_Pkg is
   task body Outputter is
   begin
      loop
         Ada.Text_IO.Put_Line("Hello");
      end loop;
   end Outputter;
end Outputter_Pkg;
```

Java's concurrency facility is based on Object-Oriented Programming. The predefined class `Thread` supplies a variety of methods relevant to specifying and controlling concurrent activities, and a user wishing to define a template for concurrently executing objects can do so by extending (subclassing) `Thread` and overriding the `run()` method. A *thread* is then an instance of such a subclass.

```
class Outputter extends Thread{
   public void run(){
      while (true){
         System.out.println("Hello");
      }
   }
}
```

The technique of subclassing `Thread` is not always applicable, however. Since Java supports only single inheritance, a class that extends `Thread` may not extend any other class. Java solves this problem by providing an interface, `Runnable`, with an (abstract) method `run()`. The Java `Thread` class implements `Runnable` by supplying a `run()` method that simply returns, and it also supplies a constructor that takes a `Runnable` parameter and produces a `Thread`. Thus an alternative style to subclassing `Thread`, and in fact one that is generally preferred, is to implement `Runnable`. Here is an example:

```
class RunnableOutputter implements Runnable{
   public void run(){
      while (true){
         System.out.println("Hello");
      }
   }
}
```

Ada provides more flexibility than Java, in several areas. First, in Ada it is straightforward to declare a single task object, a task type, or an access-to-task type. In Java, extending `Thread` is analogous to declaring an access-to-task type in Ada; Java's heap-based model means that there is no direct analog to an Ada task type or an Ada task object declaration. It is possible in Java to create a reference to a thread whose type is an anonymous `Thread` subclass, but the syntax is somewhat awkward.

Second, in Ada one may declare task objects or task types in nested scopes, with standard visibility to names in outer scopes; for example, outer variables may be updated and/or referenced. Although Java allows locally declared classes and thus permits a `Thread` subclass to be declared within a method or block, the code for the local class can only reference outer parameters and variables that are constant ("final variables" in Java parlance). Java's restriction to accessing only constants does not imply that such accesses can safely be left unsynchronized: a thread can modify the object designated by a constant of a reference type.

Ada's flexibility means that an outer scope cannot exit until all inner tasks have terminated, a condition whose checking entails a run-time price. Java has no such requirement; however, since a method with local constants may return while an inner thread that references such data is still running, the Java run-time system cannot safely use a simple stack to store method parameters and local variables. To avoid dangling references, the Java implementation must either allocate method "stackframes" on the heap, or else reserve space in a thread-specific area for a copy of all of the non-local data that the thread references.

The relationship between OOP and concurrency is a subject of ongoing research, and Java and Ada have staked out different positions. Ada's tasking support is separate from its OOP model. Allowing a (limited private) tagged type to be completed as a protected type would have added semantic and implementation complexity and was somewhat out of the scope of the language revision effort that culminated in Ada 95.

Java's thread model intrinsically uses OOP; a class that extends `Thread` or implements `Runnable` can itself be extended. However, this is less useful than it may seem. Java's thread synchronization facilities trip on the "inheritance anomaly", an interaction between synchronization and OOP that will be discussed further in §10 below.

An advantage of Java's approach is that applications needing to deal with threads in general, such as user-defined schedulers, can compose data structures with `Thread` components and methods with `Thread` parameters. In Ada one needs to use the `Task_ID` type, arguably a less direct approach. On the other hand, `Thread` is not a typical class; although `run()` is a public method of any class that extends `Thread` or implements `Runnable`, it is almost always an error to invoke `run()` explicitly.

5. TASK/THREAD CREATION AND STARTUP

5.1 Basic properties

In Ada a task is created as an effect of a task object declaration or allocation. The following example shows an allocation of an object of the task type `Outputter`:

```
with Outputter_Pkg; use Outputter_Pkg;
procedure Main is
   Ref : Outputter_Ref;
begin
   Ref := new Outputter;
end Main;
```

Ada task execution is a three-step process: create the task, activate it (i.e., elaborate its declarative part) and then, as a concurrent activity, execute the statements in the task body. The allocation of a task object entails all three steps. If a task is declared, then the declaration's elaboration creates the task object but the activation and further execution do not occur until the "begin" of the enclosing unit. In either case the execution of the task body occurs automatically.

In Java, thread execution is a two-step process, both of which are explicit in the program: construct a `Thread` object, and invoke its `start()` method. The effect of `start()` is to invoke the thread's `run()` method from a new thread of control and then return immediately.

Here is a Java version of the above Ada program:

```
class Driver1{
  public static void main( String[] args ){
    Outputter ref;
    ref = new Outputter();
    ref.start();
  }
}
```

When the `Driver1` class is loaded its main method is invoked, resulting in the declaration of the variable `ref`. The invocation of the constructor `new Outputter()` creates a new thread but does not initiate its execution. An explicit invocation of the `start()` method (inherited from `Thread`) is required, which has the effect of invoking the subclass's `run()` method from another thread of control and then immediately returning.

Alternatively, here is the version based on the `RunnableOutputter` approach; one of the `Thread` constructors takes a `Runnable` parameter:

```
class Driver2{
  public static void main( String[] args ){
    RunnableOutputter ro =
       new RunnableOutputter();
    Thread ref = new Thread(ro);
    ref.start();
  }
}
```

The main difference is Ada's automatic task activation/execution versus Java's explicit thread startup. An explicit activation mechanism was considered during the initial Ada design but was rejected because of its error-proneness (for example, forgetting to invoke it, or invoking it more than once on the same task). Java's explicit `start()` facility suffers from these problems. On the other hand, separating the creation of a task from its startup does allow certain methods to be invoked on a thread that is in the created-but-not-started state (for example, setting its "dæmon" status), and in that sense offers some additional flexibility.

5.2 Parameterization

It is sometimes convenient or necessary to pass parameters to a task/thread when it is created or started. Ada has two ways to achieve this: a discriminant of a task type, or a parameter to an entry that is accepted as the first statement in the task body. For example, suppose we want to specify the number of iterations as a parameter to an `Outputter` task. Here are the style using a discriminant:

```
package Outputter_Pkg1 is
  task type Outputter(How_Many : Natural);
  type Outputter_Ref is access Outputter;
end Outputter_Pkg1;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Outputter_Pkg1 is
  task body Outputter is
  begin
    for I in 1..How_Many loop
      Ada.Text_IO.Put_Line("Hello");
    end loop;
  end Outputter;
end Outputter_Pkg1;
```

```
with Outputter_Pkg1; use Outputter_Pkg1;
procedure Main1 is
   Ref : Outputter_Ref;
begin
   Ref := new Outputter(10);
end Main1;
```

Here is the style using a rendezvous:

```
package Outputter_Pkg2 is
  task type Outputter is
    entry Start(How_Many : in Natural);
  end Outputter;
  type Outputter_Ref is access Outputter;
end Outputter_Pkg2;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Outputter_Pkg2 is
  task body Outputter is
    How_Many : Natural;
  begin
    accept Start(How_Many : in Natural) do
       Outputter.How_Many := How_Many;
    end Start;
    for I in 1..How_Many loop
       Ada.Text_IO.Put_Line("Hello");
    end loop;
  end Outputter;
end Outputter_Pkg2;
```

```
with Outputter_Pkg2; use Outputter_Pkg2;
procedure Main2 is
   Ref : Outputter_Ref;
begin
   Ref := new Outputter;
   Ref.Start( How_Many => 10 );
end Main2;
```

In Java the declaration of class `Thread` does not allow overriding the `start()` method, and when `run()` is

overridden its signature may not be changed. Thus the Java idiom is to pass "implicit" parameters to `run()` through a constructor that stores its parameters in instance variables, which may then be referenced from `run()`.

```
public class Outputter extends Thread{
  int howMany;  // instance variable
  Outputter( int howMany ){
    this.howMany = howMany;
  }
  public void run(){
    for (int i=1; i <= howMany; i++){
       System.out.println("Hello");
    }
  }

  public static void main(String[] args){
    Outputter outp = new Outputter(10);
    outp.start();
  }
}
```

In some ways Java's constructor mechanism is more flexible than Ada's discriminants:

- A constructor may include statements that assign to static variables.
- Constructors may be overloaded to take different numbers/types of parameters.
- If a constructor stores its parameter in a (non-final) variable, the `run()` method can assign to this variable. In contrast, an Ada discriminant is a constant. (However, an access discriminant may be used to get the effect of a variable.)

On the other hand, Ada's rendezvous offers more flexibility than Java's constructors. The rendezvous has the further stylistic benefit of an explicit communication that appears at the place where it is needed, versus separating it into the two steps of passing parameters via a constructor and referencing instance variables from the `run()` method.

6. TASK/THREAD TERMINATION

6.1 Self termination

In Ada an "active" task (i.e., one that does not have any `accept` statements) completes implicitly when it reaches its "end". Since Ada allows tasks to be declared in inner scopes (including other tasks) there is a dependence hierarchy at run time.

A `terminate` alternative on a `select` statement can be used to arrange implicit termination for a "passive" task.

Analogous to an Ada task reaching its end, a Java thread will terminate when its `run()` method executes a `return` statement or reaches its end "}". Because of Java's restrictions on nesting, the language does not need to distinguish between completion and termination.

Somewhat akin to the implicit termination associated with Ada's `terminate` alternative, Java defines the concept of a "dæmon thread". A method invocation can set a thread's status to "dæmon" after the thread has been constructed but before it has been started. If all non-dæmon threads have terminated, then the Java Virtual Machine will automatically kill all dæmon threads and hence terminate the application. Such a mechanism is necessary since there are a number of threads implicitly associated with each executing application, for example the garbage collector. Requiring the programmer to explicitly terminate such threads would not be friendly.

The dæmon thread concept is somewhat more general than Ada's select-with-terminate, but with the generality comes some danger. The language rules do not specify the effect if a dæmon thread happens to be executing when it is made to terminate. If it is in the process of accessing a system resource, for example logging to a file, then the resource may be left in an inconsistent state.

Java includes a method call `t.join()` that suspends the invoking thread until thread `t` has terminated. Ada lacks a directly analogous construct, although it is simple to achieve the same effect by declaring a task in a local block.

6.2 Terminating other tasks/threads

Ada supports several ways for one task to terminate another. One technique is a "shutdown" entry that is explicitly called by one task and accepted by another. This is the cleanest style but may lead to latency depending on how the accepting task is expressed. An alternative is the `abort` statement which, if the Real-Time Annex is supported, provides a more immediate mechanism for arranging task termination while still satisfying the need for certain constructs to be "abort deferred".

Java can simulate a shutdown entry through its wait/notify mechanism described below. There is nothing that directly corresponds to the Ada `abort`, although two `Thread` instance methods, `stop()` and `destroy()`, offer some corresponding functionality. The `stop()` method is an asynchronous exception mechanism and will be described below. The `destroy()` method unconditionally and immediately kills a thread, without performing any cleanup. This is a potentially dangerous operation: if the destroyed thread was holding any monitor locks (see §7.1), then these locks will never be released and any other thread that subsequently attempts to enter one of these monitors will be deadlocked. In any event this method has not yet been implemented in the Java Development Kit.

7. MUTUAL EXCLUSION

7.1 Basic Properties

Ada supplies four mechanisms for mutual exclusion:

- A "volatile" variable, specified via a pragma, which may not be optimized into a "temporary" location such as a register
- An "atomic" variable, specified via a pragma, whose accesses are indivisible (with respect to task context switching) because of the hardware, and which may not be optimized into a "temporary" location such as a register
- A protected object/type, with associated protected operations, based on the concept of "concurrent read, exclusive write" locks and designed for efficient implementation

- A "passive" task, expressed as a loop around an accept or selective_accept statement

Java provides two techniques:

- A "volatile" variable, specified via a modifier on its declaration
- A monitor (an object with a "lock"), specified via a synchronized block or a synchronized method

Java's `volatile` modifier is equivalent to Ada's `pragma Atomic` for a standalone variable, but does not apply to components and thus is less general than Ada. Also, Java does not have the equivalent of Ada's `pragma Volatile`. The latter construct is useful when two tasks are accessing a (not necessarily atomic) data structure but are known to do so at non-overlapping times.

Java's synchronized block/method approach uses the classical monitor / critical region model (see, e.g., [12, §7]). Each object (including the "class object" associated with a class's static members) has a lock that may be thought of as a non-negative integer with an atomic test-and-set. If a thread invokes a method marked as `synchronized`, then it will be blocked unless either the method's containing object is unlocked (i.e., the lock value is 0) or the thread already holds the lock on this object. In either of these two cases the lock is incremented by 1. On return from a synchronized method the lock value is decremented by 1. (Using a count rather than a boolean for the lock allows a thread that is executing synchronized code to invoke other synchronized code on the same object.) Synchronized blocks are analogous to synchronized methods but are generally less preferable stylistically owing to the distribution of locking code throughout the program.

Whether a thread holds or releases locks when it is suspended depends on the specific circumstance. Invoking the `sleep()` method (§9.3) does not cause the release of any locks; in contrast, invoking the `wait()` method (§8.1) causes all locks to be released.

There is no notion of a queue for threads that are waiting for an object to be unlocked, and one possible implementation is for a blocked thread to be periodically awakened to test if the object has become unlocked.

7.1.2 Java Example

Here is an example of the "Readers/Writers" idiom in Java using synchronized methods:

```
class Clock{
  private int minutes=0, seconds=0;

  synchronized void tick(){
    minutes =
      (minutes + ((seconds+1)/60)) % 60;
    seconds = (seconds+1) % 60;
  }

  synchronized Clock read(){
    Clock result = new Clock();
    result.minutes=this.minutes;
    result.seconds=this.seconds;
    return result;
  }
// Selector functions to be invoked on
  // the result of read()
  int seconds(){ return this.seconds; }
  int minutes(){ return this.minutes; }
}
```

```
class Reader extends Thread{
  private Clock clock;

  Reader(Clock clock){ this.clock=clock; }

  public void run(){
    while (true){
      Clock now = clock.read();
      System.out.println("minutes: " +
                         now.minutes() );
      System.out.println("seconds: " +
                         now.seconds() );
      try {Thread.sleep(1000);}
      catch(Exception e){}
    }
  }
}
```

```
class Writer extends Thread{
  private Clock clock;

  Writer(Clock clock){ this.clock=clock; }
  public void run(){
    for (int i=1; i <= Integer.MAX_VALUE;
         i++){
      try {Thread.sleep(1000);}
      catch (Exception e){}
      clock.tick();
      System.out.println( "Tick " + i );
    }
  }
}
```

```
public class ReaderWriter{
  public static void main(String[] args){
    Clock c = new Clock();
    Writer w = new Writer(c);
    Reader r = new Reader(c);
    w.start();
    r.start();
  }
}
```

A `Writer` thread increments a `Clock` object approximately once per second. (For simplicity we ignore the cumulative drift implied by the style shown above; periodicity will be addressed below in §9.3.) The static method `sleep()` in class `Thread` suspends the calling thread for the specified number of milliseconds.

A `Reader` thread retrieves and displays the `Clock` value, also approximately once per second.

It is essential for the `tick()` and `read()` functions to be specified as synchronized, otherwise a `Reader` thread risks retrieving an inconsistent result (a new value for minutes and an old value for seconds). The "selector" functions `seconds()` and `minutes()` are not specified as synchronized, since their only purpose is to be invoked by a `Reader` thread on its copy (constructed by the `read()` method) of the shared `Clock` object. This style is

necessitated by Java's lack of "out" parameters for primitive types.

7.1.3 Ada Example

The following Ada program has the same effect as the Java version. It is somewhat simpler since Ada's "out" parameters obviate the need for selector functions. Note that an access discriminant serves the purpose of a parameterized constructor.

```
package Clock_Pkg is
  subtype Mod_60 is Integer range 0..59;

  protected type Clock is
    procedure Tick;

    procedure Read( Minutes : out Mod_60;
                    Seconds : out Mod_60 );
  private
     Minutes, Seconds : Mod_60 := 0;
  end Clock;
end Clock_Pkg;
```

```
package body Clock_Pkg is
  protected body Clock is
    procedure Tick is
    begin
      Minutes := ( Minutes +
                 ((Seconds+1)/60) ) mod 60;
      Seconds := ( Seconds + 1 ) mod 60;
    end Tick;

    procedure Read(Minutes : out Mod_60;
                   Seconds : out Mod_60) is
    begin
      Minutes := Clock.Minutes;
      Seconds := Clock.Seconds;
    end Read;
  end Clock;
end Clock_Pkg;
```

```
with Clock_Pkg, Ada.Text_IO;
use Clock_Pkg, Ada.Text_IO;
procedure Reader_Writer is
  task type Reader(C : access Clock);
  task type Writer(C : access Clock);

  task body Reader is
     Minutes, Seconds : Mod_60;
  begin
    loop
      C.Read( Minutes, Seconds );
      Put_Line( "minutes:" &
                Integer'Image(Minutes) );
      Put_Line( "seconds:" &
                Integer'Image(Seconds) );
      delay 1.0;
    end loop;
  end Reader;

  task body Writer is
  begin
    for I in Positive loop
      delay 1.0;
      C.Tick;
      Put_Line("Tick" & Integer'Image(I));
    end loop;
  end Writer;

  C : aliased Clock;
  W : Writer(C'Access);
  R : Reader(C'Access);

begin
  null;
end Reader_Writer;
```

7.1.4 Comparison of Approaches to Mutual Exclusion

Ada's protected object mechanism is more expressive, more secure, and in some places more efficient than Java's monitors:

- Ada distinguishes protected entries from protected procedures, thus making it explicit when queuing (and not simply object locking) is required.
- Protected functions in Ada may be invoked with true concurrency in a multiprocessor environment. Java lacks an equivalent facility.
- A Java class may contain both synchronized and unsynchronized methods. Thus invoking a synchronized method does not guarantee that the access will be safe. In Ada, all externally invokable operations on a protected object are protected.
- Locking an object does not imply locking the "class object" containing the static variables. A subtle programming error is for a synchronized instance method to access a static variable.
- Mutual monitor calls in Java risk deadlock (the "nested monitor" problem). In Ada, with the `Ceiling_Locking` policy supported, mutual calls across protected objects will not deadlock.
- In Ada, protected entries combine a condition test with object locking in a way that avoids race conditions. In Java, the programmer must explicitly code the condition wait/notification logic, a more error-prone approach.
- Deciding whether to specify a method as synchronized is not always easy. Unnecessarily making a method synchronized degrades performance and may lead to deadlock in the presence of mutually dependent methods. Failing to specify `synchronized` when it is needed can cause unpredictable effects.
- Java's absence of "out" parameters for primitive data makes it clumsy to express "readers/writers" programs.

In Ada, a bounded error results if a protected operation executes a construct that could potentially block. Although Java lacks such a prohibition, the language's silence on the

issue does not imply safety or soundness. If a thread blocks by calling the `wait()` method (§8.1) from synchronized code, all the thread's locks are released. The programmer needs to be careful that the each now-unlocked object's state is consistent at points where this occurs. (In fact this problem is intrinsic in Java, since `wait()` can *only* be invoked from synchronized code – a rule that avoids some race conditions.)

If a thread blocks by calling the sleep() method from synchronized code, locks are not released. This preserves object state consistency but adds time latency for those threads that attempt to invoke synchronized code on the objects.

In Ada, protected procedures and protected entries have epilog code with a resulting run-time cost [13]. Java also has some epilog code on return from a synchronized method (to decrement the lock), but the overhead is likely to be less. On the other hand, in order to simulate the effect of Ada's protected entries, a Java program needs to do explicit processing that will likely be more expensive than the Ada epilog code for barrier reevaluation.

A protected operation in Ada is "abort deferred"; enforcing this protection may add some overhead. Java has no such concept and thus avoids this expense. However, Java does pay a reliability price (see §11.1).

8. TASK/THREAD SYNCHRONIZATION AND COMMUNICATION

8.1 Synchronous control

Ada offers several features related to synchronization and communication:

- Explicit rendezvous, for general communication
- Protected entries, when access to a shared resource requires waiting for a condition
- The `Suspension_Object` type from the Real-Time Annex, for simple event-based synchronization between two tasks

Java provides one mechanism, the somewhat low-level signal-oriented methods `wait()`, `notify()`, and `notifyAll()` that are inherited from the root class `Object`. A thread calls `ref.wait()` when it needs to be suspended until the object referenced by `ref` has a particular state. Another thread, on detecting that this object's state has the desired value, invokes either `ref.notify()` or `ref.notifyAll()`. The effect of `ref.notify()` is a "pulsed signal". If any threads are suspended waiting for the referenced object, then exactly one is awakened (the language rules do not specify which one). The effect of `ref.notifyAll()` is a "broadcast signal", awakening all threads (if any) waiting for the referenced object. If no threads are suspended waiting for the referenced object, then the "signal" implied by both `ref.notify()` and `ref.notifyAll()` is ignored. Thus a call of `ref.wait()` will always unconditionally suspend the caller and release all its locks.

The bounded buffer example in the Appendix illustrates Ada's protected objects and Java's wait/notify methods.

In most respects Ada's mechanism for synchronous control is superior to Java's:

- Ada's rendezvous is a direct, explicit communication between two tasks. In Java the communicated data must be declared together with synchronization code that controls accesses to the data, a less explicit and more error-prone style.
- Java has one queue (more strictly, one "wait set") per object, whereas Ada has one queue per protected entry, implying potential overhead in Java based on the need to invoke `notifyAll()` versus `notify()`.
- The wait and notification methods are fairly low level and somewhat error prone. It is generally necessary to invoke `wait()` in a loop versus a simple conditional, since by the time a thread is awakened after a `wait()` and has reacquired the lock on the object, another thread may have reset the condition being awaited.
- There is no guarantee that the thread awakened by a `notify()` will be either the longest waiting or highest priority thread in the object's wait set.
- Java's synchronization logic – calls on `wait()`, `notify()`, and `notifyAll()` – is embedded in method bodies. With an Ada protected entry, the wait condition is much more easily identifiable syntactically. Separating the barrier from the algorithmic code is important in coping with the Inheritance Anomaly (see §10 below).
- Java requires that a thread invoking any of the wait/notification methods hold the lock on the target object (a run-time test), or an exception will be thrown. (This requirement avoids a race condition in which one thread calls `obj.wait()`, but before it is entered in `obj`'s wait set another thread invokes `obj.notify()` or `obj.notifyAll()`. By the time the first thread is entered in `obj`'s wait set it has missed the signal.) One effect of the Java rules is that a thread invoking `notify()` or `notifyAll()` does not thereby release the object's lock. The awakened thread(s) must compete with the notifying thread, and with each other in the case of `notifyAll()`, introducing both run-time overhead and the potential for a race condition. The Ada approach does not suffer from these problems.
- For simple synchronization, Ada's `Suspension_Object` is likely to yield highly optimized performance. Java lacks such a construct, and it is much harder for an implementation to optimize objects used as the target of `wait()`.
- A program that uses `notify()` to awaken a waiting thread may need to be modified to use `notifyAll()` as a side effect of a maintenance change. More generally, it is not always easy to see when one needs to use `notifyAll()` instead of `notify()`.

Since synchronization code in Java is contained in method bodies, it can depend on parameters to the method. Although Ada does not allow entry barriers to refer to parameters (an efficiency-based restriction), the requeue statement allows a protected entry to interrogate its parameter(s) and then suspend if necessary.

8.2 Asynchronous control

Ada provides several mechanisms

- the `Hold` / `Continue` facility from the Real-Time Annex
- the asynchronous select statement

Java's `suspend()` / `resume()` methods are similar to Ada's `Hold` / `Resume`; however, these methods have been deprecated in Java 1.2. The problem is that if a thread is the target of a `suspend()` then it does not release any monitor locks. If the thread that is to perform the `resume()` needs to acquire one of the locks held by the suspended thread, then the two threads will be deadlocked. This problem does not arise in Ada, since `Hold` is deferred until the target task completes execution of a protected operation.

Java does not have a facility directly analogous to Ada's asynchronous select.

Java provides an `interrupt()` method allowing one thread to set a `boolean` flag asynchronously in another thread, which the latter can check or poll. Invoking `interrupt()` on a thread suspended by `join()`, `sleep()` or `wait()` is supposed to awaken the thread by throwing an `InterruptedException`. These exception-throwing effects are not yet implemented.

In Ada, the flag-setting functionality of `interrupt()` can be achieved through a `Boolean` variable marked as `pragma Atomic` and visible to the two tasks. A side effect of Java's `interrupt()` method is a notational inconvenience: an exception handler is needed for `InterruptedException` when `run()` invokes common methods such as `sleep()` and `join()`, since an instance of this exception class will be thrown if `interrupt()` is called on a suspended thread.

9. REAL-TIME SUPPORT

A language intended for real-time programming needs to ensure time and space predictability and also needs to supply specific features for scheduling and priorities, time manipulation, and low-level support. This section compares Ada and Java with respect to these criteria.

9.1 Time and space predictability

A real-time application by its very nature needs to perform processing within time constraints; in many situations a late answer that is completely correct is less desirable than an on-time answer that is less accurate. In order to analyze a system design and to know whether all deadlines can be met, a software developer needs to know the maximal required time for any operation, and also needs to ensure that the program does not exhaust the run-time data store. As general-purpose, high-level languages, both Ada and Java provide features that interfere with these goals. For example, returning from a subprogram in Ada entails performing finalization of local controlled objects, and waiting for inner tasks to terminate. In Java, garbage collection can occur at unpredictable times and take an unbounded amount of time. In both languages it is possible for an application to run out of space: the "activation record" stack may overflow, the heap may become full, or the heap may become fragmented.

Although on the surface the two languages seem to suffer from similar problems, in fact the issues are much more tractable in Ada. A major element of Ada's design philosophy was that unused generality not exact a run-time price, and Ada was intended from the start to be applicable in the hard real-time domain. Ada 95's `pragma Restrictions` has an assortment of arguments that the programmer can specify to indicate features that are not being used, and the implementation is expected to exploit this information through more efficient compiled code and simpler run-time support. Moreover, since Ada is a stack-based language, it is feasible to compose realistic programs that either do not use the heap at all, or else do all their heap allocations during package elaboration.

These principles have been borne out in practice. Compiler vendors such as Aonix have implemented restricted-feature profiles certified in accordance with stringent safety-critical standards, where time and space predictability are of obvious importance. Even tasking may be available; recent work on the Ravenscar [14] profile has proposed a set of tasking features that can be used in efficient, high-integrity applications with predictable timing properties.

In contrast, time and space predictability in Java is much more difficult to accomplish. Java is intrinsically a heap-based language; only primitive data and references go on the stack. Pre-allocating all objects at system start-up, though feasible in principle, will lead to a rather constrained programming style. Enhancing the language to allow stack-based objects, or allocations from stack-based pools, would be significant changes. Implementing certifiable, predictable-performance garbage collection in an environment that requires hard real-time response remains a goal rather than an accepted state of the practice for Java.

9.2 Scheduling and priority

Ada's Real-Time Annex defines a number of mechanisms:

- Subtypes for priority ranges, and subprograms to dynamically set and retrieve a task's priority
- Semantic distinctions between a task's *active* and *base* priorities
- Pragmas that dictate policies for dispatching, locking, and entry queuing

Java defines a priority range, 1 through 10, and provides methods to set and retrieve a thread's priority. However, the treatment of priorities is implementation dependent, perhaps surprising in light of Java's well-publicized goal of software portability. If the underlying operating system uses time-sliced scheduling, then a low priority thread may run even though a higher priority thread is eligible for execution.

The number of priority levels in Java is 10, which is insufficient in many real-time applications. Ada's Real-Time Annex requires at least 30 values, a more comfortable limit.

In Ada, a task can invoke the scheduler through the idiom "`delay 0.0;`" for example to implement cooperative

multiplexing for equal-priority tasks. Java has an analogous mechanism, the `yield()` method, but without the guarantee that the invoking thread goes to the tail of the ready queue.

9.3 Timing-related features

Ada supplies several facilities dealing with time:

- Predefined packages `Ada.Calendar` and `Ada.Real_Time`, with subprograms to retrieve and manipulate time values
- Relative and absolute `delay` statements
- Time out on an entry call, accept statement, selective_accept, and asynchronous select

Here is a simple periodic task in Ada; the types `Time_Span` and `Time`, and the function `Clock`, are declared in `Ada.Real_Time`, a package provided by the Real-Time Annex.

```
task type Periodic(Period:
                     access Time_Span);
task body Periodic is
  Next_Time : Time := Clock;
begin
  loop
    ... -- Perform action
    Next_Time := Next_Time + Period.all;
    delay until Next_Time;
  end loop;
end Periodic;
```

Java likewise has several relevant time-related features:

- Predefined classes and methods to retrieve and manipulate time values
- Relative delay (the `sleep()` method)
- Timeout on `wait()` and other methods

Here is a Java version of the periodic task:

```
class Periodic extends Thread{
  protected final int period;
  Periodic(int period){this.period=period;}
  public void run(){
    long nextTime =
      System.currentTimeMillis();
    while (true) {
      ... // Perform processing
      nextTime += period;
      int delay = (int)( nextTime -
              System.currentTimeMillis() );
      try {Thread.sleep( delay );}
      catch (InterruptedException e) {}
    }
  }
}
```

Analogous to an Ada discriminant, the constant (final variable) `period` is not initialized as part of its declaration, but rather is set, and cannot thereafter be changed, as an effect of the constructor.

The Java version is somewhat more verbose than Ada, in part because `run()` needs to handle the `InterruptedException` that might be thrown by `sleep()`. More importantly, the relative delay of Java's `sleep()` method runs the risk of missing deadlines. If a `Periodic` thread is preempted by a higher priority thread after the delay is computed but before the `sleep()` invocation has taken effect, then the delay that will eventually be used will not have taken the preemption time in account. This phenomenon was observed in Ada 83 with the "relative" delay statement, and is the reason that the "absolute" delay mechanism was introduced into Ada 95.

Although Java supplies overloaded versions of `sleep()` and other time-related methods, each taking a parameter that specifies the number of nanoseconds in addition to the number of milliseconds, there are no assurances about the granularity of the clock used in their implementation. In contrast, Ada 95 imposes requirements both in the core language (no coarser than 20 milliseconds) and in the Real-Time Annex (50 microseconds).

9.4 Low-level facilities

Ada has made a conscious effort to provide access to the underlying hardware environment. Chapter 13 of the Ada Reference Manual defines a large number of features for interrogating and/or controlling aspects of representation and for interfacing with hardware. The interfacing pragmas offer a well-defined and portable approach to integrating Ada with foreign code, and this includes assembler language modules. The Systems Programming Annex adds requirements on the implementation and its documentation, in effect ensuring that whatever can be done in machine language is accessible through Ada. Although many of the facilities will by their nature be target environment dependent (for example the interrupt handling semantics), the language still provides an overall framework.

Java provides almost no facilities for low-level manipulation. This is hardly surprising; for several reasons. First, Java was initially targeted to the JVM, which is at a higher semantic plane than the underlying hardware. Performing low-level processing is generally unnecessary and undesirable in this environment. Second, one of the fundamental types for low-level programming is the address, a concept that meshes poorly with Java's semantics and its run-time garbage collector. Although Java supplies "native methods" to get to the underlying hardware, these are not as well defined as Ada's interfacing facilities, and programmers need to be quite familiar with the implementation details in order to use native methods safely and effectively.

9.5 Work in progress on real-time Java

Java's problems with respect to real-time programming are widely acknowledged even in the Java community itself, and several efforts are underway to attempt to address these concerns. One of the early proposals in this area is NewMonics' PERC (Portable Executive for Reliable Control) [15], a product and technology comprising several main elements:

- Enhancements to the Java language: the `timed` block (which times out, releasing all locks, if the block's execution time exceeds the specified duration) and an `atomic` block (which must be executed either to completion or not at all).

- Enhancements to the API comprising two principal packages: `RealTime`, which relates to budgeting time and memory, and `Embedded`, which supports I/O ports, interrupt vectors, and persistent memory.
- Enhancements to the Java Virtual Machine, including an implementation of a garbage collection algorithm that meets the requirements for real-time applications.
- Tools for determining resource requirements both analytically and empirically

Spurred by the PERC effort and other real-time Java activities, the U.S. Government's National Institute for Standards and Technology (NIST) organized a working group to look into these issues. This has led to two independent efforts, one under the auspices of Sun's Java Community Process, and the other from a consortium of companies interested in Java for real-time applications but not interested in the terms of Sun's JCP. The two groups are expected to release their specifications for real-time extensions to the Java platform during Summer 2000.

10. THE "INHERITANCE ANOMALY"

Over the past several years programming language researchers have been investigating the interaction between concurrency and OOP known as the "inheritance anomaly". The issue concerns inheriting from a class whose methods contain *synchronization code*: constructs that enforce mutual exclusion, or that control thread suspension/resumption based on condition checks/signals. As observed by [16, p. 108], "synchronization code cannot be effectively inherited without non-trivial class re-definitions". Depending on the language's synchronization scheme, the subclass author may need to know the implementation of methods in the superclass, and the introduction of new methods in the subclass may necessitate overriding seemingly-unrelated superclass methods. These effects compromise encapsulation.

The inheritance anomaly relates to three topics: *mutual exclusion*, *condition-based execution*[1], and *scheduling* The second term refers to a situation where a thread invoking a method may need to be blocked until an enabling condition becomes true. How this condition and its signaling are expressed depends on the language features. Here are the main issues surrounding the inheritance anomaly:

- Some or all of the superclass's methods may need to be executed with mutual exclusion, and likewise for the subclass's methods. How easy is it for the programmer to express such requirements? If two objects' methods call each other, how is deadlock to be prevented?
- Whether a method call is to be immediately executed, or whether the calling task/thread needs to be blocked, may be based on run-time conditions. Bloom [17], [18] identified a number of such constraints, which can be categorized as follows:
  - *Local state*. A condition may be a function of the values of instance variables; e.g. whether a bounded buffer is empty, full, or partially-filled dictates whether only a "put" method, only a "get" method, or either, may be invoked.
  - *History information*. A condition may depend on the history of previous calls; e.g., in a "reader/writer", a "read" operation should be blocked until after at least one "write" method has been performed.
  - *Request parameters*. A condition may depend on parameters passed to the method; e.g., whether a "chopstick guardian" method in a Dining Philosopher's solution is invokable depends on which chopsticks are requested [19].

  In light of these factors, what are the implications of defining a subclass where either some of its methods, or some of the superclass's methods, or both, exhibit condition-based execution? In particular, if a subclass method imposes new conditions, is it necessary to override the implementation of superclass methods?
- If several threads/tasks are competing for an object that needs to be accessed with mutual exclusion, or if a condition (or conditions) awaited by multiple threads/tasks has become true, then the choice of which thread to run will depend on the scheduling policy. Bloom identified the following scheduling policy aspects:
  - *Type of request*. It may be necessary to give preference to some methods over others. For example, in a "readers/writers" program, priority might be given to callers of "write" to ensure that readers obtain fresh data.
  - *Order of requests*. For different applications, different policies may be appropriate for scheduling their method invocations: FIFO, priority-based, non-deterministic, etc.

  How can the class author specify the appropriate scheduling policy, and how is this affected by inheritance?

An in-depth treatment of the inheritance anomaly is beyond the scope of this paper; indeed, the topic remains the subject of ongoing research. More comprehensive discussion may be found in [16] (which coined the term "inheritance anomaly"), [18], and [20]. We here summarize how Ada and Java compare with respect to the questions posed above.

10.1 The Inheritance Anomaly in Ada

10.1.1 Mutual Exclusion

Ada's tagged type mechanism offers OOP semantics, but without mutual exclusion or synchronization control. Ada's protected type mechanism offers mutual exclusion and

[1] The first two issues could be unified, since mutually exclusive access to an object is a special case of condition-based execution where the condition is the absence of other threads executing a method on the object in question. However, since both Ada and Java offer different mechanisms for the two effects, and since the implementation of mutual exclusion and condition-based execution can be quite different, it is useful to treat them separately.

synchronization control, but without the OOP semantics for inheritance. As a consequence, combining OOP with mutual exclusion requires a "mix-in" style. [5, §13.2] summarize three main techniques:

- The root type contains no protection logic; synchronization is added by the descendant type's implementation.
- The root type contains protection logic, for example an encapsulated "mutex" component in the type's implementation, which is called from class-wide operations to lock the object before a dispatching call, and to unlock it afterwards. This technique is also described in [21].
- Synchronization is centralized in a protected type declared in a root package. The data are passed through an access discriminant using an access-to-class-wide type, and are thus decoupled from the protection logic. The protected operations make dispatching calls on primitive operations.

None of these solutions is ideal, and working around the drawbacks leads to somewhat complicated styles. The third approach seems to be the safest, but the inability to extend a protected type gives this solution a non-OO flavor. If a new class is added to the inheritance hierarchy, with new operations, then the root protected type needs to be changed.

10.1.2 Condition-Based Execution

[18] observed that "inheritance anomalies are directly attributed to the failure to consider certain classes of Bloom's constraints on synchronization" and reviewed a number of languages against these criteria. (Java was not in this set, possibly because it was still a rather new language at the time the paper was written.) Ada 95's protected type facility came closest to satisfying the criteria.

- Local state can be tracked through protected components, which are referenced from entry barrier conditions disjoint from the entry body. (Separating the "guard" logic from the algorithmic code is important in avoiding tedious and error-prone changes during inheritance.)
- History can likewise be tracked with protected components.
- Although a barrier cannot reference an entry parameter, the equivalent effect may be obtained through a requeue.

The main issue, as with mutual exclusion, is the lack of extensibility for protected types. [21] discuss the notion of "synchronization compatibility" between a subclass and a superclass – for example, a subclass cannot weaken the guards on an operation, and the subclass states must be a partitioning of the superclass states. They propose a solution based on encapsulating the protection logic in a "mutex" in the root type. A drawback is that the resulting style seems somewhat heavy.

10.1.3 Scheduling

With an implementation that supports the Real-Time Annex, the user has control over a number of elements of the scheduling policy including, with some restrictions based on efficiency, the dispatching and queuing policies. In particular, giving precedence to particular operations may be achieved dynamically through entry barriers (checking the `'Count` attribute) or statically through the `Priority_Queuing` argument of pragma `Queuing_Policy`. This pragma more generally establishes whether requests will be served FIFO, priority-based, or through some other policy.

As with mutual exclusion and condition-based execution, these mechanisms are separate from the OOP features. Indeed, OOP has generally not been widely used in the real-time community, and it is not surprising that the scheduling features of the Real-Time Annex should aim more for efficiency than for blissful harmony with OOP.

10.2 The Inheritance Anomaly in Java

10.2.1 Mutual Exclusion

If a synchronized method is inherited, it is also synchronized in the subclass. One of the effects of Java's thread model being based on OOP is thus a simple approach to inheriting mutual exclusion protection. However, Java still suffers from several problems:

- Mutually dependent methods may lead to deadlock. This is a special case of the "nested monitor" issue described earlier, but it can easily arise through a combination of several common OOP idioms: "passing the buck", where a subclass method invokes the corresponding superclass method, and "nested dispatching" where a superclass method dynamically binds to another method on the same object.
- If a synchronized method is overridden, then `synchronized` must be specified explicitly on the overriding declaration. Otherwise the subclass's method will not be synchronized, a perhaps subtle error.

10.2.2 Condition-Based Execution

Java's condition checks (a "while" loop on `wait()`) and state transitions (invocations of `notify()` and `notifyAll()`) are buried in algorithmic code. As observed by [16] and [18], specifying synchronization in code bodies makes the effect of inheritance anomalies much worse than in languages where these are separated, for example with conditions given by "guards". It is almost inevitable that a Java subclass that requires additional synchronization logic will need to completely re-implement superclass methods containing wait/notification calls, thus breaking encapsulation.

One of Java's objectives is to minimize the "binary compatibility" problem [2, §13]. If a new version of a class `K` is produced, for example with new functionality or a revised implementation, a user who had defined a subclass `K1` of the original class should be able simply to relink with the new version. If either version of `K` contains synchronization code, this will probably not work. The user will need the source code for the new version and may need to modify the source code for the subclass `K1`.

10.2.3 Scheduling

As noted earlier, Java leaves the semantics of scheduling and priorities completely unspecified. The effect depends on the implementation, and the class author has no control over scheduling policy.

10.3 Summary of Inheritance Anomaly

The inheritance anomaly is a difficult problem, and neither Ada nor Java offer completely satisfactory solutions. Ada's main shortcoming is the inability to extend a protected type, a restriction that was motivated by efficiency considerations and concern about semantic complexity. Since one of the main purposes of Ada tasking was to support real-time applications with minimal overhead, the generality of extendable protected types was not considered a sufficiently large gain in expressability to offset the costs.

Java integrates thread support with OOP, and its inheritance of synchronized methods addresses some of the issues of the inheritance anomaly. However, its immediate susceptibility to "nested monitor" deadlocks, and the need for explicit low-level synchronization through calls on `wait()` or `notify()`/`notifyAll()` are intrinsic problems with the language design.

In summary, Ada's concurrency model is a much more appropriate starting point than Java's for addressing the inheritance anomaly. A possible approach, for a future version of the language, would be to add OOP semantics to protected types. One proposal for such a scheme is given in [22].

11. INTERACTION WITH EXCEPTION HANDLING

11.1 Synchronous versus asynchronous exceptions

One of the axioms of Ada semantics is that exceptions are always synchronous. In a pre-standard draft of the language in the early 1980's an asynchronous exception was included, the `'Failure` attribute; raising `T'Failure` would halt execution of `T` and transfer control to an exception handler included in `T`'s body, thereupon allowing `T` to perform necessary cleanup. This feature was removed before the language was standardized, since it was difficult to specify semantics for the construct that the programmer could depend upon in writing reliable code. For example, raising `T'Failure` while `T` is accessing a file could leave the file in an inconsistent state.

Interestingly, Java seems to be repeating Ada's history. Through Version 1.1, Java included asynchronous exceptions; in particular, invoking `t.stop()` on a thread `t` would throw an instance of the exception class `ThreadDeath`, and an alternate version of `stop()` would allow the invoker to specify an arbitrary exception object to be thrown in the target thread. However, this method is deprecated in Java 1.2. There is no guarantee where the target thread is executing when the exception is thrown, and thus it is possible for either program data or run-time system data to be in an inconsistent state. Invoking `stop()` also has the effect of releasing all locks held by the stopped thread. Other threads will be able to enter "aborted" monitors, and they may see inconsistent data.

Ada does not ignore the issue of asynchronous communication, but rather than using the exception mechanism it provides a limited form of asynchronous control transfer based on either a time out or the acceptance of an entry call. This construct, the asynchronous select statement, provides some guarantees on which operations are "critical sections" ("abort deferred" in Ada parlance) with respect to such asynchronous events.

11.2 Propagating exceptions out of tasks/threads

As a consequence of Ada's "no asynchronous exceptions" principle, an unhandled exception raised in a task body's statement part is not propagated; the task dies silently. (Propagating the exception to, say, the unit that activated the task would raise an asynchronous exception.) To prevent "silent task death" syndrome, a programmer can include in the task body an exception handler with an "others" choice to rendezvous with an error-logging task.

Since the signature of the `run()` method does not include a `throws` clause, a Java thread can never propagate an instance of a checked exception class. (A side effect is that if `run()` invokes a method with a `throws` clause, the invocation must be in a `try` block with a `catch` clause covering the specified exception class.) However, it is possible for a thread to propagate an unchecked exception such as `ArrayIndexOutOfRangeException`. In such a situation the exception is passed as a parameter (of class `Throwable`) to the method `uncaughtException`. This method's default effect is to print out a stack trace based on the exception, except when the parameter is an instance of `ThreadDeath` in which case `uncaughtException` simply returns.

The two languages provide roughly the same effects. One difference is that Java mandates the printing of a stack trace on the propagation of uncaught exceptions (other than for `ThreadDeath`). Such functionality is useful in an interactive environment but of questionable value where the program is running on an embedded processor.

11.3 Exceptions propagated from task/thread-specific constructs

In Ada an exception propagated from a protected operation is propagated to the caller; an exception propagated from an accept statement is propagated both within the called task and to the caller. Other tasking constructs raise exceptions based on specific circumstances. For example, calling an entry of a terminated task raises `Tasking_Error` at the call; raising an exception during elaboration of a task's declarative part (i.e. during activation) propagates `Tasking_Error` to the unit that caused the task activation. Note that in this latter case the exception is synchronous, since the activating unit is suspended while the new task is being activated.

In Java an exception may be propagated from synchronized blocks or methods; in such a situation the lock count is decreased as an effect of the propagation.

12. OTHER ISSUES

Several other differences between Ada and Java are as follows

- Java supplies the concept of a "thread group", allowing the programmer to define a collection of threads that may be manipulated all at once by certain methods. This concept is useful in Java's security model for applets. Ada has no corresponding construct.
- Ada's Systems Programming Annex uses the tasking model, in particular the protected object, as the basis for interrupt handling. Java has no mechanism for interrupt handling.

13. CONCLUSIONS

Ada and Java have taken quite different approaches to supporting concurrency. Ada provides a general, high-level model based on explicit communication (the rendezvous), and a structured approach to mutual exclusion (protected objects), while also supplying lower-level mechanisms and specific scheduling semantics and control that may be needed for real-time and other applications. In contrast, Java's approach relies on the classical monitor construct for mutual exclusion, and "pulsed" / "broadcast" signals for thread synchronization and communication. Java is thus susceptible to the well-known "nested monitor" deadlock problem, and its signal-oriented approach to communication is error-prone and may entail additional context switches. Moreover, the absence of semantics for priority means that thread scheduling is completely implementation dependent.

Although certain elements of Java's thread model provide functionality not found in Ada (for example, thread groups), on the whole Ada's approach to concurrency is more reliable and more portable than Java's.

In the real-time area, Java is at a number of disadvantages compared to Ada, both in general with respect to the goals of predictable performance and, specifically, in connection with dynamic allocation, garbage collection, and thread scheduling. Enhancing Java to be more appropriate for real-time applications currently is an area of active interest.

Feature	Ada	Java
Concurrency unit	task	thread
General types for concurrency	`Task_ID`	`Thread` `ThreadGroup`
Creation	declaration or allocation	construction (allocation)
Parameterization	discriminant, entry	constructor
Startup	implicit after activation	explicit via `start()`
Termination (implicit)	select statement with terminate	dæmon thread
Termination (explicit)	`abort`	`stop()` `destroy()`
Wait for termination	block declaring a local task	`join()`
Mutual Exclusion	`Atomic` variable	`volatile` variable
	protected operation	synchronized method
	critical region controlled by semaphore	synchronized block
Synchronization and communication	entry barrier on protected object	`wait()`
	`Suspension_-Object`	`wait()`
	implicit barrier reevaluation at end of protected procedure or protected entry	`notify()` `notifyAll()`
	rendezvous	no corresponding feature
Asynchronous control	`Hold` `Continue`	`suspend()` `resume()`
	asynchronous select statement	no corresponding feature
	`Atomic` variable	`interrupt()`
Time	`delay 0.0`	`yield()`
	`delay N`	`sleep(1000*N)`
	`delay until T`	no corresponding feature
	Time-out on entry call, accept	Time-out on `wait()`
Scheduling policy	User-specifable	Implementation-defined

APPENDIX A: BOUNDED BUFFER EXAMPLE

This Appendix presents Ada and Java versions of the classical "bounded buffer" idiom. The Ada version is a generic package where the element type is a formal generic parameter; the Java version uses `Object` as the element type.

A "producer" inserts an element into a buffer by invoking the buffer's "`put`" method. A "consumer" removes an element from a buffer, in FIFO fashion, by invoking the buffer's "`get`" method. A producer is blocked when the buffer is full, and a consumer is blocked when the buffer is empty.

In both versions, the producer runs at twice the frequency of the consumer, so that eventually the buffer will be full.

A.1 Ada Version

```
generic
  type Element is private;
package Generic_Buffer_Pkg is
  type Element_Array is
    array (Positive range <> ) of Element;
  protected type Buffer(Max : Natural) is
    entry Put( Item : in Element ) ;
    entry Get( Item : out Element );
  private
    Data : Element_Array(1..Max);
    Next_in, Next_Out : Integer := 1;
    Count             : Natural := 0;
  end Buffer;
end Generic_Buffer_Pkg;
```

```
package body Generic_Buffer_Pkg is
  protected body Buffer is
    entry Put( Item : in Element )
      when Count < Max is
    begin
      Data(Next_In) := Item;
      Next_In       := (Next_In mod Max)+1;
      Count         := Count+1;
     end Put;

     entry Get( Item: out Element )
       when Count > 0 is
     begin
       Item     := Data(Next_Out);
       Next_Out := (Next_Out mod Max)+1;
       Count    := Count-1;
     end Get;
  end Buffer;
end Generic_Buffer_Pkg;
```

```
with Generic_Buffer_Pkg;
procedure Producer_Consumer is
  package Int_Buffer_Pkg is
    new Generic_Buffer_Pkg( Integer );
  use Int_Buffer_Pkg;

  Buff : Buffer(20) ;

  task Producer;
  task body Producer is
  begin
    for J in 1..100 loop
      delay 0.5;
      Buff.Put( J );
    end loop;
  end Producer;

  task Consumer;
  task body Consumer is
    K : Integer;
  begin
    for J in 1..100 loop
        Buff.Get( K );
        delay 1.0;
    end loop;
  end Consumer;
begin
  null; -- wait for tasks to terminate
end Producer_Consumer;
```

A.2 Java Version

The following Java version illustrates both a `Thread` subclass and a `Runnable` implementation. Recall that `protected` has quite different meanings in Ada and Java.

Both the `Producer` and `Consumer` threads need to operate on a common `Buffer` object. As is typical in Java, a reference to this object is passed as a parameter to the constructor for the two classes, stored in an instance variable, and accessed from the `run()` methods.

```
class Buffer{
  protected final int max;
  protected final Object[] data;
  protected int nextIn=0, nextOut=0,
            count=0;

  public Buffer( int max ){
    this.max  = max;
    this.data = new Object[max];
  }
  public synchronized void put(Object item)
  throws InterruptedException{
    while (count == max) { wait(); }
    data[nextIn] = item;
    nextIn       = (nextIn+1) % max;
    count++;
    notify(); //a waiting consumer, if any
  }

  public synchronized Object get()
  throws InterruptedException{
    while (count == 0) { wait(); }
    Object result = data[nextOut];
    nextOut       = (nextOut+1) % max;
    count--;
    notify(); // a waiting producer, if any
    return result;
  }
}
```

```
class Producer implements Runnable{
  protected final Buffer buffer;
  Producer( Buffer buffer ){
    this.buffer=buffer;
  }

  public void run() {
    try{
      for (int j=1; j<=100; j++){
        Thread.sleep( 500 );
        buffer.put( new Integer(j) );
      }
    }catch (InterruptedException e)
    {return;}
  }
}
```

```
class Consumer extends Thread{
  protected final Buffer buffer;
  public Consumer( Buffer buffer ){
    this.buffer = buffer;
  }
  public void run(){
    try {
      for (int j=1; j<=100; j++){
        Integer p = (Integer)(buffer.get());
        int k = p.intValue();
        Thread.sleep( 1000 );
      }
    } catch (InterruptedException e)
      {return;}
  }
}
```

```
public class ProducerConsumer{
  static Buffer buffer = new Buffer(20);
  public static void main(String[] args){
    Producer p  = new Producer(buffer);
    Thread   pt = new Thread(p);
    Consumer c  = new Consumer(buffer);
    pt.start();
    c.start();
  }
}
```

The `put` and `get` methods each include a `throws` clause specifying `InterruptedException`, since the `wait` method in class `Thread` includes such a clause and `put` and `get` do not catch exceptions from this class. However, the implementation of each `run` method needs to catch `InterruptedException`; `get`, `put`, and `sleep()` can propagate exceptions from this class, and the `run()` method, lacking a throws clause, must catch any exceptions propagated from methods that it invokes. It would not have been possible to simply put a `throws InterruptedException` clause on the implementation of the `run()` methods, since the signature of this method in the superclass `Thread` and the interface `Runnable` lack a `throws` clause. As noted earlier, a method is not allowed to throw more kinds of exceptions than are specified in the `throws` clause of the version of the method that it is overriding.

In this example the signals sent at the end of `put()` and `get()` can simply be `notify()` versus `notifyAll()`. Even if there were many `Producer` and `Consumer` threads, versus just one of each as above, there is no way that both a `Producer` and a `Consumer` could be in the wait set for a given `Buffer` object. (If a `Consumer` thread is in the `Buffer` object's wait set, that is because the `Buffer` is empty. A `Producer` thread may attempt to call `put()` while the object is locked, but that does not place the caller in the object's wait set. Since all the threads in the wait set are waiting for the same condition, awakening all of them is of no benefit: after one `Consumer` finishes executing `get()` the `Buffer` will again be empty and all the other `Consumer` threads will suspend immediately on re-evaluating the condition. Thus a simple `notify()`, awakening just one thread in the wait set, is sufficient. Analogous reasoning applies to the choice of `notify()` versus `notifyAll()` at the end of `get()`.)

A.3 Comparison of Styles

The Ada version is more readable, for several reasons

- Synchronization is clearly expressed through entry barriers as opposed to Java's `wait()` / `notify()`.
- The exception handling code in the Java version is extraneous and distracts from the main processing.
- Generic units are more type-safe than Java's approach with `Object` as the buffer element type, and they do not incur the run-time overhead and stylistic heaviness of the casts between Integer and `Object`.

The Ada program also has some efficiency advantages

- In Ada the `Buffer` object will go on the stack; there is no need for heap management or an additional level of indirection on referencing the buffer data. In Java the `Buffer` will be constructed on the heap, with resulting run-time costs.

The main Java advantage is the flexibility of its dynamic thread model, but this was not really exploited in the example.

REFERENCES

[1] Intermetrics, Inc.; *Ada Reference Manual - Language and Standard Libraries*, ISO/IEC 8652:1995.

[2] J. Gosling, B. Joy, and G. Steele; *The Java™ Language Specification*, Addison-Wesley, 1996.

[3] K. Arnold and J. Gosling, *The Java™ Programming Language (2nd edition)*; Addison-Wesley, 1998.

[4] B. Brosgol; "A Comparison of the Object-Oriented Features of Ada 95 and Java™", *Proc. Tri-Ada '97*, ACM SIGAda, 1997.

[5] A. Burns and A. Wellings; *Concurrency in Ada*, Cambridge University Press, 1995.

[6] D. Lea; *Concurrent Programming in Java™ - Design Principles and Patterns*, Addison-Wesley, 1997

[7] S. Oaks and H. Wong; *Java Threads*, O'Reilly & Associates, 1997.

[8] D. Flanagan, *Java in a Nutshell* (2nd edition); O'Reilly & Assoc., 1997.

[9] B. Brosgol, “A Comparison of Ada and Java as a Foundation Teaching Language”, ACM SIGAda *Ada Letters*, Sept.-Oct. 1998.

[10] Sun; “The Java™ Language; and Overview”, `http:\\java.sun.com/docs/overview/java/java-overview-1.html`

[11] ANSI/IEEE *IEEE Standard for Binary Floating-Point Arithmetic*; ANSI/IEE Std. 754-1985; 1985.

[12] A. Burns and G. Davies; *Concurrent Programming*, Addison-Wesley; 1993.

[13] E. Giering and T. Baker; “Implementing Ada Protected Objects - Interface Issues and Optimization”, *Proc. Tri-Ada ’95*, ACM SIGAda, 1995.

[14] A. Burns, B. Dobbing, and G. Romanski, “The Ravenscar Profile for High-Integrity Real-Time Programs”, in *Reliable Software Technologies - Ada Europe ’98* (L. Asplund, ed.); Springer-Verlag Lecture Notes in Computer Science #1411; 1998.

[15] K. Nilsen and S. Lee, *PERC™ Real-Time API* (Draft 1.3); NewMonics Report, 1998.

[16] S. Matsuoka and A. Yonezawa; “Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in *Research Directions in Concurrent Object-Oriented Programming*; MIT Press, 1993.

[17] T. Bloom, “Evaluating Synchronization Mechanisms”, in *Proc. of the Seventh ACM Symposium on OS Principles*; 1979.

[18] S. Mitchell and A. Wellings; “Synchronization, Concurrent Object-Oriented Programming, and the Inheritance Anomaly”, in *Comput. Lang.*, Vol. 22, No. 1, 1996

[19] B. Brosgol, “The Dining Philosophers in Ada 95”, in *Proc. 1996 Ada-Europe International Conference*, Springer Lecture Notes in Computer Science 1088; 1996.

[20] L. M. J. Bergmans, *Composing Concurrent Objects*, Ph.D. Thesis, University of Twente (Netherlands); 1994.

[21] G. Schumacher and W. Nebel; “How to Avoid the Inheritance Anomaly in Ada”, in *Proc. 1998 Ada-Europe International Conference*, Springer Lecture Notes in Computer Science 1411; 1998.

[22] O.P. Kiddle and A.J. Wellings, “Extensible Protected Types in Ada – EPT”, in *Proc. SIGAda ’98*; ACM SIGAda, 1998.