THE JR PROGRAMMING LANGUAGE: CONCURRENT PROGRAMMING IN AN EXTENDED JAVA

RONALD A. OLSSON
Department of Computer Science
University of California, Davis

AARON W. KEEN
Department of Computer Science
California Polytechnic State University

Kluwer Academic Publishers
Boston/Dordrecht/London

Chapter 1

INTRODUCTION

Concurrent programming is concerned with writing programs having multiple processes that may execute in parallel. The topic originated in the 1960s when the invention of independent device controllers (channels) led people to organize operating systems as concurrent programs, even for single-processor machines. Since then, rapid developments in computer architecture have led to an increasingly large number of multiprocessor architectures, such as shared-memory multiprocessors, multicomputers, and networks of workstations. The operating systems for these architectures are all instances of concurrent programs. More importantly, multiprocessor architectures make it possible to write application programs that exploit the concurrency inherent in the hardware. Both distributed systems, multiprocessor systems, and hybrids (e.g., distributed systems that include some multiprocessors) are prevalent today and they are likely to remain so.

A concurrent program specifies two or more processes that cooperate in performing a task. Each process consists of a sequential program. The processes cooperate by communicating, which in turn gives rise to the need for synchronization. Communication and synchronization are programmed by reading and writing shared variables or by sending and receiving messages. Shared variables are most appropriate for concurrent programs that execute on a single processor or a shared-memory multiprocessor. Message passing is most appropriate for distributed programs that execute on multicomputers or networks of workstations. (Message passing can also be used on shared-memory machines.)

This book describes the JR programming language and shows how it can be used to write concurrent programs for a variety of hardware architectures and software applications. JR is an extension of the Java programming lan-

guage [28] with additional concurrency mechanisms based on those in the SR (Synchronizing Resources) programming language [6, 9].

Java has proven to be a clean and simple (and popular) language for object-oriented programming. Even so, the standard Java concurrency model is rather limited. It provides threads, a primitive monitor-like mechanism, and remote method invocation (RMI). Although these features are useful, they offer little flexibility in the design and implementation of concurrent programs.

JR provides a richer and more flexible concurrent programming model than Java. JR adapts the following features from SR: dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, asynchronous message passing, semaphores, and shared variables. JR takes a novel object-oriented approach to synchronization whereas SR is not object-oriented.

Thus, JR inherits and extends one of SR's distinguishing attributes: its expressive power. The communication and synchronization mechanisms listed above include most of the ones that have proven popular and useful. This makes JR suitable for writing concurrent programs for *both* shared- and distributed-memory applications and machines.

In addition to being expressive, JR is easy to learn and use for someone who has some background with Java. Its variety of concurrent programming mechanisms is based on only a few underlying concepts. Moreover, these concepts are generalizations of ones that have been found useful in sequential programs. The concurrent programming mechanisms are also integrated with the sequential ones, so that similar things are expressed in similar ways. An important design goal has been to retain the "feel" of Java while providing a richer concurrency model.

Part I of this book describes the concurrent aspects of JR in detail and gives numerous, smaller examples. Part II develops complete programs for several larger applications: matrix multiplication, partial differential equations, the traveling salesman problem, a distributed file system, and discrete event simulation. These illustrate the use of JR for distributed programming using message passing and parallel programming using shared variables. JR is implemented on top of Java, so, in principle, it can run on any platform that supports Java, including networks of workstations and shared-memory multiprocessors. JR programs can also be executed on single processor machines, in which case process execution is interleaved. The current JR implementation runs on UNIX-based (Linux, Mac OS X, and Solaris) and Windows-based systems.

The remainder of this chapter gives a brief overview of JR. First we describe the main components of the language. Then we present complete programs that solve several familiar problems. The solutions illustrate the structure of JR programs and some—but by no means all—of the language's power and flexibility. Finally, we describe how to create and execute JR programs.

1.1 Key JR Components

As noted above, JR extends Java with additional mechanisms for supporting concurrency. The key new features are virtual machines, remote objects, and operations.

A JR virtual machine represents an address space, which is located entirely on one physical machine. These virtual machines can be created dynamically during program execution in a way similar to how objects are created. JR virtual machines can be "populated" with remote objects, which are essentially the usual instances of classes. In JR, a remote object is simply a Java object that has been created in a way slightly different from the usual Java `new`. Thus, JR object creation is dynamic, as in Java. A class in Java serves as the unit of compilation and encapsulation; a class in JR serves a similar role. A JR class may contain anything that a Java class may contain plus it may contain additional JR features. The one difference in the use of classes is that in JR all classes must be compiled together.

One such feature is the process, which represents a separate thread of control.[1] JR provides a process abbreviation. Processes can be created dynamically and can share variables in the same object, in the same class (static variables), and in other classes on the same virtual machine (public static variables). Processes can also communicate and synchronize by means of operations.

An operation can be considered a generalization of a method: It has a name and can take parameters and return a result. An operation can be invoked in two ways: synchronously by means of a call statement or asynchronously by means of a send statement. An operation can also be serviced in two ways: by a method or by input statements. These ways of servicing an operation support local and remote method calls and rendezvous. As we shall see in Part I, this variety of possibilities provides a great deal of flexibility and power for solving concurrent programming problems.

JR contains several mechanisms that are abbreviations for common uses of operations; these can be used to simplify many programs. Abbreviations include process declarations, op-method declarations, receive statements, and semaphores. JR also provides a few additional statements that are useful for concurrent programming. The reply and forward statements provide additional ways to use operations.

JR also provides a means to deal with program quiescence. A JR program becomes *quiescent* when all of its processes have terminated or deadlocked. At that point, the JR implementation will normally terminate the program's execution. Instead, however, JR allows an operation to be registered as the "qui-

[1] JR uses the traditional term "process" to represent this abstraction. As we will see in later chapters, JR processes are actually mapped to Java threads. To further confuse matters, the term "process" is often used to represent an operating system process, which might contain multiple threads of execution.

escence operation"; this operation will be invoked when the program becomes quiescent. This feature is useful to avoid having to write code to determine when processes have terminated.

JR programs can use all of the many packages provided for Java. For example, these include common math functions and a variety of input/output functions. JR programs can also interact with Java packages for building GUIs (graphical user interfaces), such as AWT and Swing; Chapter 20 show some examples of such interaction.

1.2 Two Simple Examples

One of the best ways to learn a new programming language is to start writing programs. To do so, it helps to look at examples.

A standard first example in a programming language text is a program that writes the message "Hello World!" on the standard output file. In JR, the following program does the trick:

```
import edu.ucdavis.jr.JR;
public class HelloWorld {
  public static void main(String [] args) {
    System.out.println("Hello World!");
  }
}
```

It is nearly identical to the equivalent program in Java. The first difference is that JR programs must import the JR package. However, to save space, most examples in this book will omit that line; be sure to include it in any programs that you actually try to compile, though! For the same reason, our code in this book generally does not check for errors in input data or command-line arguments. The second difference is that the JR program's main method must appear in a `public` class.

As noted earlier, the sequential aspects of JR are identical to those of Java (with the exception of one extension seen in Chapter 3). However, JR provides extensions to Java to simplify the writing of concurrent programs, as the next example illustrates.

This program uses two processes to perform two independent computations:

```
public class TwoProcesses {
  private static final int [] A = { 8,  4, 11, 19};
  private static final int [] B = {14, 17,  9,  3};
  private static final int N = A.length;
  private static process p1 {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      sum += A[i];
    }
```

```
    System.out.println("sum of A is " + sum);
  }
  private static process p2 {
    int sum = 0;
    for (int i = 0; i < N; i++) {
      sum += A[i] * B[i];
    }
    System.out.println("inner product of A with B is " + sum);
  }
  public static void main(String [] args) {
  }
}
```

Process `p1` computes the sum of the elements in array `A` and outputs the result; process `p2` computes the inner product of the elements in array `A` with those in array `B` and outputs the result.

This program illustrates four important aspects of JR. Chapter 4 discusses these aspects in detail.

The first aspect to note is that JR programs use the same scoping as Java programs. Consequently, each process gets its own copy of variables declared local to it (such as `sum` and `i`), but the processes *share* variables and constants (such as `A` and `B`) declared at the class level.

In this program, since the processes only read shared constants, there is no potential for both processes updating a shared variable at about the same time and interfering with each other in doing so. Such a *race condition* (or *data race*) can occur with shared variables. An example illustrating a race condition is given in Section 4.1. Processes can use synchronization to protect access to shared variables. One such technique is demonstrated in Section 1.5. Others are demonstrated in subsequent chapters; e.g., see Section 5.5 for an example of how to use only shared variables to program synchronization and see Section 6.1 for an example of how to use semaphores.

The second important aspect of JR illustrated by the `TwoProcesses` program involves the program's output. It outputs two lines, one from each process, but the order in which the lines appear is non-deterministic. The output might be `p1`'s output followed by `p2`'s output, or vice versa. Which ordering occurs depends on the order in which the two processes execute, which is also non-deterministic.

The third aspect illustrated by the `TwoProcesses` program is that the processes were declared to be `static`. Non-static processes are also allowed, but static processes are slightly simpler to use, so we use them in many of the examples in this book.

The final aspect deals with program termination. As noted in Section 1.1, a JR program terminates when all of its processes have terminated or deadlocked; it will also terminate when it has executed a `JR.exit`.

1.3 Matrix Multiplication

Now consider the problem of multiplying two `N` $\times$ `N` real matrices `A` and `B`. We first present a sequential program to solve this problem and then show how to modify the program to compute all $\texttt{N}^2$ inner products in parallel.

The following program first reads in the source matrices, then computes the matrix product, and finally prints the result matrix. (The code omits the details of reading in the matrices as that code just uses standard Java features.) The main method reads in the arrays, instantiates a `MMMultiplier` object to do the actual computation, and then invokes the `print` method in that object.

```
public class MMMain {
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...
    MMMultiplier m = new MMMultiplier(A, B, N);
    m.print();
  }
}


public class MMMultiplier {
  int N; // A and B are NxN
  double [][] C;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.N = N;
    C = new double [N][N];
    // compute NxN inner products
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        C[r][c] = 0.0;
        for (int k = 0; k < N; k++) {
          C[r][c]+= A[r][k] * B[k][c];
        }
      }
    }
  }
  public void print() {
    // output C
    for (int r = 0; r < N; r++) {
      for (int c = 0; c < N; c++) {
        System.out.print(C[r][c]+" ");
      }
      System.out.println();
    }
  }
}
```

The code in `MMMultiplier`'s constructor computes n^2 inner products using nested for statements. The inner for statement computes the inner product of row `r` of `A` and column `c` of `B` and stores the result in `C[r][c]`. The code in the `print` method prints matrix `C`, with each row printed on a separate line.

Since the inner products are independent of each other, we can compute all N^2 in parallel, as shown below. This program will not be very efficient, since each process does very little computation, but we could readily modify it to use fewer processes (see Exercise 1.2 and also Chapter 15). The main class the same as the previous main class, except it uses a quiescence operation to print the result, as described later below.

```
public class MMMain {
  private static MMMultiplier m;
  public static void main(String [] args) {
    int N;  // A and B are NxN
    double [][] A, B;
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (edu.ucdavis.jr.QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    m.print();
  }
}
```

The `MMMultiplier` code now performs the matrix multiplication by using `compute` processes.

```
public class MMMultiplier {
  int N; // A and B are NxN
  double [][] A, B, C;
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;
    C = new double [N][N];
  }
  process compute ( (int r = 0; r < N; r++),
                    (int c = 0; c < N; c++) ) {
    // compute the inner product for C[r,c]
    C[r][c] = 0.0;
    for (int k = 0; k < N; k++) {
      C[r][c]+= A[r][k] * B[k][c];
    }
```

```
  }
  public void print() {
    // output C --- code same as before
  }
}
```

The heading on `compute` contains two quantifiers, so $\texttt{N}^2$ processes are created, one for each combination of values for `r` and `c`. In fact, `r` and `c` are parameters to each instance of `compute` and are available in `compute`'s body. Each process computes one inner product, just as each iteration of the innermost loop does in the sequential program. The `compute` processes are created at the end of execution of `MMMultiplier`'s constructor.

When inner products are computed in parallel, `C` should not be printed out until all processes have terminated. As mentioned in Section 1.1, a program may register a quiescence operation, which is invoked when JR has detected that the program has finished computation and is about to terminate. Hence, the code associated with the quiescence operation is executed *after* the rest of the computation terminates. In the program above, the main method registers `done` as the program's quiescence operation. Once the `compute` processes terminate, the code in `done` is executed to print out `C`. By using a quiescence operation, we do not need to add synchronization code to the rest of the program to determine when all the `compute` processes have terminated. This feature of JR makes many programs, including this one, easy to write. Chapter 15 describes how to structure solutions to this problem in ways that do not require using a quiescence operation.

1.4 Concurrent File Search

The programs given so far are very short, so they consist of a single class. Often it is best to employ multiple classes. The last two examples in this chapter illustrate how to do so.

The `grep` family of UNIX commands is commonly used to search for patterns in files. For example, the following command searches each of the named files:

```
grep string filename1 filename2 ...
```

Each line containing `string` is printed on standard output. (If there is more than one file, each line of output begins with the name of the file.) The `grep` command searches each file sequentially.

The following JR program gives a simplified, concurrent implementation of the above command. In particular, it searches the files in parallel, one process for each file. The program has the same arguments as `grep` above: a pattern string and one or more file names. (It does not implement the `grep` command's other useful features, such as searching for strings matching patterns specified by regular expressions; see Exercise 1.5.) Like `grep`, the program prints all

lines that contain the pattern string on the standard output. A string containing the file name concatenated with a colon is printed at the front of each line. Since searching and printing proceed in parallel, however, lines from different files will be interleaved.

The program consists of two classes. Execution begins in the `grepmain` class, which creates a `grepworker` object for each filename given on the command line.

```
public class GrepMain {
  public static void main(String [] args) {
    if (args.length < 2) {
      System.err.println(
        "needs arguments: pattern filename {filename}");
      JR.exit(1);
    }
    String pattern = args[0];
    // create a GrepWorker object for each filename
    for (int k = 1; k < args.length; k++) {
      new GrepWorker(pattern, args[k]);
    }
  }
}
```

The constructor for class `grepworker` has two parameters: `pattern` and `filename`. It saves the parameters into object variables. When the constructor is done executing, the `new` in `grepmain` completes and an instance of process `search` in the newly instantiated `grepworker` object is created implicitly. The `search` process finds all instances of `pattern` in `filename` and writes them out; the file name and a colon are printed at the front of each line.

```
import java.io.*;
public class GrepWorker {
  String pattern, filename;
  public GrepWorker(String pattern, String filename) {
    this.pattern = pattern;
    this.filename = filename;
  }
  private process search {
    try {
      FileReader fr = new FileReader(filename);
      BufferedReader br = new BufferedReader(fr);
      String line;
      while ((line = br.readLine()) != null) { // get null on EOF
        if (line.indexOf(pattern) >= 0) {
          System.out.println(filename + ":" + line);
        }
      }
      fr.close();
```

```
    } catch (FileNotFoundException fe) {
      System.err.println("can't open " + filename);
    } catch (IOException ioe) {
      System.err.println("IO Exception for " + filename);
    }
  }
}
```

All objects in the above program execute on the same machine. However, we can readily modify the program so that different instances of `grepworker` execute on potentially different machines. For example, suppose a file name is specified on the command line as `machine:filename`. Also, suppose that `main` separates `machine` from `filename` and stores the values in string variables with those names. Then `main` can create a `grepworker` object on `machine` by executing

```
vm vmcap;
vmcap = new vm() on machine;
new remote GrepWorker(pattern, filename) on vmcap;
```

A `vm` in JR is a virtual machine (address space). The first line declares a reference for a `vm`. The second line creates a new `vm` on the machine whose name is stored in variable `machine`. The third line creates an instance of `grepworker` on the newly created `vm`, and hence on a potentially remote machine (as indicated by the `remote` keyword). The effect of making the above changes is that each `grep` object will open `filename` on the machine on which it is executing. (This program assumes that, for reasons explained in Section 10.8, the names of the files to be searched are specified as relative to home directory or are specified as absolute pathnames on the remote machine.)

1.5 Critical Section Simulation

As a final example, we present a program that illustrates a few of the numerous message-passing mechanisms available in JR. The program also illustrates how one can construct a simple simulation of a solution to a synchronization problem.

The following program contains `numusers` instances of a `user` process, each of which repeatedly executes a critical section of code and then a non-critical section. At most one process at a time is permitted to execute its critical section. If more than one process wants to enter its critical section at the same time, the one with the highest priority is permitted to do so. Each `user` process has an index `i`; the lower the index value, the higher the priority of the process. We simulate the duration of critical and non-critical sections of code by having each `user` process "nap" for a random number of milliseconds.

```
import java.util.Random;
public class CSS {
```

```
  private static op void CSenter(int);
  private static op void CSexit();
  private static process arbitrator {
    while (true) {
      inni void CSenter(int id) by id {
        System.out.println("user " + id + " in its CS at " +
                           System.currentTimeMillis());
      }
      receive CSexit();
    }
  }
  private static final int numusers = 3, rounds = 4;
  public static void main(String [] args) {
  }
  private static process user( (int i = 1; i <= numusers; i++) ) {
    Random r = new Random();  // seed with system time
    for (int j = 1; j <= rounds; j++) {
      call CSenter(i);                // enter critical section
      try {
        Thread.sleep(r.nextInt(100));  // delay up to 100 msec
      } catch (Exception e) {e.printStackTrace();}
      send CSexit();                  // exit critical section
      try {
        Thread.sleep(r.nextInt(1000)); // delay up to 1 second
      } catch (Exception e) {e.printStackTrace();}
    }
  }
}
```

The `CSS` class contains an `arbitrator` process that implements two operations: `CSenter` and `CSexit`. It first uses an input statement (`inni`) to wait for an invocation of `CSenter`. This is JR's rendezvous mechanism. If there is more than one invocation of `CSenter`, the one that has the smallest value for parameter `id` is selected, and a message is then printed. Next the `arbitrator` uses a receive statement to wait for an invocation of `CSexit`. Receive is a special case of `inni` that can be used when one just needs to receive a message or, in this case, simply a signal.

Each `user` process calls the `CSenter` operation to get permission to enter its critical section, passing its index `i` as an argument. After "napping" the process then invokes the `CSexit` operation. The `CSenter` operation must be invoked by a synchronous call statement because the `user` process has to wait to get permission. However, since a `user` process does not need to delay when leaving its critical section, it invokes the `CSexit` operation by means of the asynchronous send statement.

The program employs several methods in Java packages. The `System.currentTimeMillis` method in the print statement returns the number of milliseconds since a particular epoch. The `Thread.sleep` method causes

a process to "nap" for the number of milliseconds specified by its argument. The `nextInt` method in the `Random` class returns a pseudo-random integer between 0 and its argument.

1.6 Translating and Executing JR Programs

To execute a JR program, one must first create one or more files containing the program text. The names of these files must end with `.jr`. Following Java requirements, the JR class `x` must be placed in the file `x.jr` if `x` is public. For example, the "Hello world" program must be placed in a file named `HelloWorld.jr`.

The standard tool to translate and execute a JR program is `jr`. Assuming the directory containing `jr` is in a particular user's search path, the user can compile and execute the program in `HelloWorld.jr` by using the command

```
jr HelloWorld
```

Note that the `.jr` suffix does *not* appear in this command; only the name of the class containing the main method does. The `jr` command assumes that all `.jr` files in the current directory are part of the program. After the main class name, additional arguments to `jr` are the command-line arguments to be passed to the main method.

The `jr` command performs several actions. Assuming no errors, it invokes each of the following:

- the JR compiler (`jrc`) to generate Java code for each `.jr` file (it also generates additional Java classes as needed by the program);
- the Java compiler to translate that code to bytecode;
- the RMI compiler to adapt the translated code to execute with RMI (which JR uses to distribute programs); and
- the JVM (Java Virtual Machine) to run the translated bytecode.

The `jr` command creates in the current directory a new subdirectory named `jrGen` (first deleting the old one if it already exists). It uses this directory for all the files created by the above steps, e.g., `.java` and `.class` files. The programmer should have no need to be concerned with the contents of these files but also should not modify them. However, `jrGen` is needed to run a program, so it should not be deleted by the user if the program is to be executed several times.

Several other JR tools provide flexibility in applying the above steps. The following table summarizes these tools:

`jr`	translates and executes a JR program
`jrc`	translates a JR program
`jr_rmic`	adapts JR-translated Java code to execute with RMI
`jrrun`	executes an already-translated JR program
`jrgo`	like `jr`, but tries to determine the name of main class
`jrgox`	like `jrrun`, but tries to determine the name of main class

See Appendix C for further details on developing and executing JR programs.

1.7 Vocabulary and Notation

We begin by explaining the notation and conventions we will be using in the remainder of the book. As already seen in this chapter, we typeset JR syntactic tokens and programs in the Courier (typewriter) typeface.

JR's syntax extends Java's syntax with additional statements and forms of declarations and expressions; these extensions introduce several new keywords. The specific keywords and syntax will be introduced as we describe the various language mechanisms. The exact syntax and the complete set of keywords is given in Appendix A.

We will present the syntax of the JR extensions in a form in which each syntax display conveys what an element of the JR grammar looks like in a program. For example, consider how we might describe the syntax of a simplified version of JR's receive statement. (Chapter 7 gives the full description.) A receive statement names an operation and gives a list of zero or more variables separated by commas. It has the following general form:

`receive` *op_id* `(` *variable*`,` *variable*`, ...` `)`

The keyword `receive`, the parentheses, and the commas are JR tokens, so they are typeset in Courier. The items *op_id* and *variable* are non-terminals in the JR grammar. When an item such as *variable* can be repeated, we will always list two instances and two separators and follow them with an ellipsis. We will also say whether there must be zero or more or one or more instances of the item.

We will when possible follow common Java terminology in presenting JR syntax. The key syntactic items we will use are summarized in the following table.

block	a block of zero or more statements enclosed within `{` and `}`
expr	an expression
id	an identifier
variable	a variable

Exercises

⇒ As noted in the Preface, source code for all programming examples and the "given" parts of the programming exercises are available on the JR webpage.

1.1 Execute the `TwoProcesses` program several times to see whether the order of output differs between executions. If not, then add an invocation of `Thread.sleep` to force the other order of output.

1.2 Add to the `TwoProcesses` program a third process, which is to find the maximum element in both of the arrays.

1.3 (a) Compare the execution times of the sequential and parallel matrix multiplication programs for various size matrices. Which is more efficient?

(b) Modify the parallel program so that it uses only `N` processes, each of which computes one row of result matrix `C`. Compare the performance of this program to your answers to part (a).

1.4 (a) Execute the concurrent file search program using different patterns and files on a UNIX system. Compare the output to that of the `grep` command. Now try piping the output of your JR program through the `sort` command, and compare the output to that of `grep`. What happens if the file-name arguments to your JR program are given in alphabetical order?

(b) Modify the program to create instances of `grep` on different machines, as described in Section 1.4. Experiment with this version of the program.

1.5 Modify the concurrent file search program so that it allows the search string to be a regular expression. To save yourself a lot of work, use an existing Java regular expression package like `gnu.regexp`.

1.6 Execute the critical section simulation program several times and examine the results. Also experiment with different nap intervals by modifying the argument to the `nextInt` method. Modify the program by deleting the phrase `by id` in the `arbitrator` process, and execute this version of the program several times. How do the results compare to that of the original program? What if `by id` is replaced by `by -id`?