Generic Operations and Capabilities in the JR Concurrent Programming Language

Hiu Ning (Angela) Chan[†], Andrew J. Gallagher[†],
Appu S. Goundan[†], Yi Lin William Au Yeung[†],
Aaron W. Keen[‡], and Ronald A. Olsson[†§]

[†]Department of Computer Science
University of California, Davis, CA 95616 USA

[‡]Computer Science Department
California Polytechnic State University
San Luis Obispo, CA 93407 USA

[§]Olsson is the corresponding author:
olsson@cs.ucdavis.edu
+1 530-752-7004 (office)
+1 530-752-4767 (fax)

May 20, 2008

Abstract

The JR concurrent programming language extends Java with additional concurrency mechanisms, which are built upon JR's operations and capabilities. JR operations generalize methods in how they can be invoked and serviced. JR capabilities act as reference to operations. Recent changes to the Java language and implementation, especially generics, necessitated corresponding changes to the JR language and implementation. This paper describes the new JR language features (known as JR2) of generic operations and generic capabilities. These new features posed some interesting implementation challenges. The paper describes our initial implementation (JR21) of generic operations and capabilities, which works in many, but not all, cases. It then describes the approach our improved implementation (JR24) uses to fully implement generic operations and capabilities. The paper also describes the benchmarks used to assess the compilation and execution time performances of JR21 and JR24. The JR24 implementation reduces compilation times, mainly due to reducing the number of files generated during JR program translation, without noticeably impacting execution times.

Keywords: concurrency, JR operations, Java, generics, language design, language implementation

1 Introduction

The JR concurrent programming language [1, 2] extends Java to provide a rich concurrency model, based on that of the SR concurrent programming language [3]. JR performs synchronization using an object-oriented approach, and provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation.

JR provides much of this added functionality via its *operation* abstraction, which can be considered a generalization of a method and which can be invoked and serviced in different ways. Along with operations, JR provides *capabilities*, which act as (first-class) references or pointers to operations and which are useful in concurrent programs to connect together the different participants in the computation.

During program compilation, JR source programs are translated into standard Java programs, which are then compiled using the standard Java compiler. During program execution, JR programs use the JR runtime support system, which helps provide JR's extended functionality. Both the JR language translator and the JR runtime support system are written in standard Java.

The original JR language, which we call JR1, was an extension of Java 1.x. Its implementation originated from the implementation of Java 1.2.

Java 5.0 (or 1.5)[1] [4] made significant additions to Java 1.x. Java 5.0 contains several new language features including generics, enhanced for-loop, autoboxing/unboxing, typesafe enums, varargs, static import, and annotations [5]. Generics allows a type or method to operate on objects of various types while providing compile-time safety. It adds compile-time safety to Java's Collections Framework and eliminates the drudgery of casting [5]. The implementation of Java also changed for Java 5.0, including a redesign and restructuring of the translator and in how it handles RMI. (The latest version of Java is 6.0 (or 1.6). The differences between Java 5.0 and Java 6.0 are not significant for the focus of this paper.)

To accommodate these changes in Java, we made corresponding changes to the JR language and implementation [6]. We use JR2 to refer to this new version of JR. The most interesting and challenging of the changes is generics,

[1] Both version numbers "1.5.0" and "5.0" can be used to identify this release. Version "5.0" is the product version, while "1.5.0" is the developer version.

specifically how we added generic operations and generic capabilities. The design aspects were straightforward, but the implementation aspects were not. The implementation of JR2 is based on the implementation of Java 5.0. That is, we re-implemented JR, merging our changes for JR1 (and adapting those changes to fit within the framework of the Java 5.0 implementation) and adding new code for the new JR2 features.

This paper describes our experience with adding generic operations and generic capabilities to JR. Our initial approach extended our implementation of JR1, the most recent version being JR 1.00061. This initial implementation of JR2 was designated JR 2.00001 [6]; we refer to it as JR21. Although this approach works in most cases, it does not work in some important cases involving capabilities for generic operations. We devised an alternate implementation, which works in all cases. This implementation of JR2 is designated JR 2.00004.[2]

This new approach involves a fundamental change in the code that the JR translator generates. In JR21, each operation is represented as its own class with the operation's arguments represented exactly as declared. In JR24, all operations are represented as one predefined class with the operations' arguments represented as a single array of Java Objects. This new approach has the benefit of significantly reducing the amount of code produced for each JR program; specifically, it reduces the number of files produced for JR operations and capabilities. Thus, it reduces compilation times yet produces execution times that are nearly the same.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the JR language. Section 3 describes the new JR2 language features of generic operations and generic capabilities. Section 4 discusses the JR2 implementation in general; how JR21 implements generic operations and capabilities, which works in many, but not all, cases; and the approach the JR24 implementation uses to fully implement generic operations and capabilities. Section 5 compares the compilation and execution time performances of JR21 and JR24; as noted earlier, JR24 reduces compilation times without impacting execution times. Section 6 explores some related issues and

[2]Version 2.00002 was the first version using this new approach. Versions 2.00003 and 2.00004 contain several bug fixes and other enhancements. Version 2.00006 is the latest release (January 2008); it corrects several bugs in JR24. None of the bug fixes has a significant effect on the performance results described in this paper. Version 2.00005 was an experimental version described in [6]. It was based on JR21 and predates 2.00003 and 2.00004; it was a first step toward JR24, but it was not a released version of JR.

presents some related work. Finally, Section 7 concludes.

2 Overview of the JR Concurrent Programming Language

JR extends Java with a richer concurrency model [2, 7], based on that of SR [8, 3]. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, rendezvous, asynchronous message passing, semaphores, and shared variables.

Of specific interest in this paper are JR's operations and capabilities, which are used to effect some of the above features.

An operation can be considered a generalization of a method. Like a method, it has a name and can take parameters and return a result. Like a method declaration, an operation declaration specifies the *signature* of the operation, i.e., the return type and the types of parameters. Unlike a method, an operation can be invoked and serviced in different ways, which yields flexibility in solving concurrent programming problems.

More specifically, an operation can be invoked in two ways: synchronously by means of a call statement(`call`) or asynchronously by means of a send statement (`send`). An operation can also be serviced in two ways: by a method or by input statements (`inni`). This yields the following four combinations (from [2]):

Invocation	*Service*	*Effect*
`call`	method	procedure (method) call (possibly remote)
`call`	`inni`	rendezvous
`send`	method	dynamic process creation
`send`	`inni`	asynchronous message passing

JR allows several abbreviations for common uses of operations: process declarations, op-method declarations, receive statements, and semaphores. JR also provides a few additional statements that involve operations and that are useful for concurrent programming: the reply, forward, and concurrent invocation statements.

As an example, the program in Figure 1 (from [2]) illustrates a simple client-server model. The program uses the `process` abbreviation to specify that `N` client processes and one server process are to be created when

```
public class Model {
  private static final int N = 20; // number of client processes
  private static op double request(int, char);

  private static process client( (int i = 0; i < N; i++) ) {
    ...
    double d;
    d = request(i, 'w');
    ...
  }
  private static process server {
    while (true) {
      inni double request(int id, char data) {
        // handle request; put answer in ans
        double ans;
        ...
        return ans;
      }
    }
  }
  public static void main(String [] args) {
  }
}
```

Figure 1: Simple client-server JR program.

the program begins execution. Specifically, the `process` declaration for the clients uses a *quantifier*, which is similar to the control expressions in a `for` statement, to specify `N` client processes, each with its own value of `i`. Each client invokes the server via the `request` operation. This invocation, because it appears within an expression, is a call invocation; i.e., the client waits for the server to return a value. The server handles one invocation of `request` on each iteration of its loop; it does so using an `inni` statement.

JR provides *capabilities*, which act as (first-class) references or pointers to operations and which are useful in concurrent programs, to connect together the different participants in the computation. The declaration of a capability specifies the signature of the operations to which it can refer.

To illustrate capabilities, Figure 2 (from [2]) shows how Figure 1 can be rewritten. The `request` operation no longer returns a value. Instead, it now takes an additional parameter: a capability for an operation to which it should send the result it computes. The server receives that capability

```
public class Model3 {
  private static final int N = 20; // number of client processes
  private static op void request(cap void (double), char);

  private static process client( (int i = 0; i < N; i++) ) {
    op void results(double);
    ...
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    ...
  }
  private static process server {
    while (true) {
      cap void (double) results_cap;  char data;  double ans;
      receive request(results_cap, data);
      // handle request; put answer in ans
      ...
      send results_cap(ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

Figure 2: Simple client-server JR program (Figure 1) rewritten using capabilities and send/receive.

in local variable `results_cap` and uses it to send back results to the operation to which the capability points. The `inni` statement in the server has been replaced by a receive/send pair. Each client declares a local operation, `results`, whose parameterization is that of the result messages. It passes a capability for that operation as the first parameter to `request`. The call invocation of `request` in the client has been replaced by a send/receive pair. The new program is equivalent to the old one, except a client can perform other work between when it sends its invocation of `request` and when it receives its result.

As another example, the code in Figures 3 and 4 (from [2]) represents a server, in which each request is handled by a separate process, and clients. The server passes back a capability to the client that allows the client to

```
public class Server {
  public op cap void (String) startup(int n) {
    op void line (String);
    reply line;
    for (int k = 0; k < n; k++) {
      String s;
      receive line(s);
      System.out.println(s);
    }
  }
}
```

Figure 3: Server class for conversation continuity example.

interact with its server process via a private conversation, i.e., conversational continuity [9]. This interaction is accomplished by having the server process execute reply, passing back a capability for its local operation. (Like a return statement, a reply statement returns a value; unlike a return statement, a reply statement causes the executing process to continue execution after the reply statement.) Specifically, each invocation of `startup` creates a new process to handle the request; that process passes back a capability for its local operation `line`. The operation `line` takes a String as its parameter; accordingly, the signature of the capability that operation `startup` returns specifies a String parameter. The client sends the server `n` messages as the conversation.

3 Generics: A New JR2 feature

The main new language feature for JR2 is generics. In addition to supporting generics wherever Java supports generics (e.g., generic classes), JR2 also supports, as new JR language features, generic operations and generic capabilities.

3.1 Generic Operations

Generic operations in JR are similar to generic methods in Java. Figures 5 and 6 show a simple JR program with a generic operation `get()`. Class `MyShape` contains a single member variable `id` of type `E`, where `E` is a pa-

```
public class Client {
  private static Server server = new Server();
  private static process client( (int i = 1; i <= 10; i++) ) {
    final int N = 5;
    String t[] = new String [N];
    for (int x = 0; x < N; x++) {
      t[x] = i + "hi";
    }
    cap void (String) c = server.startup(N);
    for (int k = 0; k < N; k++) {
      send c(t[k]);
    }
  }
  public static void main(String [] args) {
  }
}
```

Figure 4: Client class for conversation continuity example.

rameterized type provided to the class `MyShape`. Operation `get()` returns a value whose type is `E`. For the program in Figure 6, that type is `String`, and the program outputs `"square"`, `"triangle"`, and `"rectangle"`.

3.2 Generic Capabilities

Capabilities in JR can now be generic. As a simple example, the code in Figures 5 and 6 can be rewritten to use a generic capability. The revised class `MyShape`, shown in Figure 7, now declares `gcap`, which is parameterized

```
public class MyShape <E> {
    protected E id;
    public MyShape (E id) {
        this.id = id;
    }
    public op E get() {
        return id;
    }
}
```

Figure 5: Example of a JR class with a generic operation, `get`.

```
import java.util.ArrayList;

public class GenericsOpMain {
    public static void main(String [] args) {
        ArrayList<MyShape> shapes = new ArrayList<MyShape>();

        shapes.add(new MyShape<String>("square"));
        shapes.add(new MyShape<String>("triangle"));
        shapes.add(new MyShape<String>("rectangle"));

        for (MyShape<String> item: shapes) {
            System.out.println(item.get());
        }
    }
}
```

Figure 6: Example JR code using a generic operation.

by `E`, the class's parameterized type. The using program now invokes `get()` indirectly via `gcap`; Figure 8 shows the changed invocation.

As another example, consider generalizing the code in Figures 3 and 4 by making it generic. This new server code appears in Figure 9. Note how the capability returned by `startup` and the operation `line` are parameterized by the server class's type parameter `E`. The only change to the client code is in how it declares and creates the server, as seen in Figure 10.

As a more complicated example using generic capabilities, Figure 11 shows a class with a capability, `mycap`. `mycap`'s return type and single parameter are each a generic capability for an operation with signature `E(E)`, i.e., for an operation that returns `E` and takes a single parameter `E`, where `E` is the class's parameterized type. This kind of operation parameterization occurs in practice, for example, when N processes attempt to "pair up" with each other. Each process has a generic operation, say `D`, that is to be made known to exactly one other of the N processes. Each such process sends a (generic) capability for its `D` to an operation, `exchange`, serviced by a coordinator process, and receives back a (generic) capability for its partner's `D`. The signature of the `exchange` operation is similar to the signature of the operation in Figure 11.

```
public class MyShape <E> {
    protected E id;
    public cap E () gcap;
    public MyShape (E id) {
        this.id = id;
        this.gcap = get;
    }
    public op E get() {
        return id;
    }
}
```

Figure 7: Example of a JR class with a generic capability, `gcap`.

```
System.out.println(item.gcap());
```

Figure 8: Example JR code using a generic capability. (The rest of Figure 6 remains the same.)

```
public class Server<E> {
  public op cap void (E) startup(int n) {
    op void line (E);
    reply line;
    for (int k = 0; k < n; k++) {
      E s;
      receive line(s);
      System.out.println(s);
    }
  }
}
```

Figure 9: Server class for generic conversation continuity example.

```
  private static Server<String> server = new Server<String>();
```

Figure 10: Client class creation of generic Server. (The rest of Figure 4 remains the same.)

```
public class GenericCap<E> {

    public cap cap E(E) (cap E(E)) mycap =
        new op cap E(E) (cap E(E));
    process p {
        while (true) {
            inni cap E(E) mycap (cap E(E) x) {
                System.out.println("in mycap");
                return noop;
            }
        }
    }

    public static void main (String [] args) {
        GenericCap<String> gc = new GenericCap<String>();
        cap String(String) test = null;
        test = gc.mycap(test);
    }
}
```

Figure 11: Example of a capability with generics.

4 Implementation

This section first gives an overview of the implementations of JR, focusing especially on operations and capabilities. It then describes the JR21 and JR24 implementations of generic operations and capabilities.

4.1 Overview

4.1.1 General Approach

During program compilation, JR source programs are translated into standard Java programs, which are then compiled using the standard Java compiler. During program execution, JR programs use the JR runtime support system, which helps provide JR's extended functionality. Both the JR language translator and the JR runtime support system are written in standard Java. The JR runtime environment also includes instances of the JR virtual machine (JRVM), one instance for each extant VM in the execution of the JR program. The JRVM provides services in addition to those of the underlying Java virtual machine. (Thus, JR programs ultimately run on standard Java

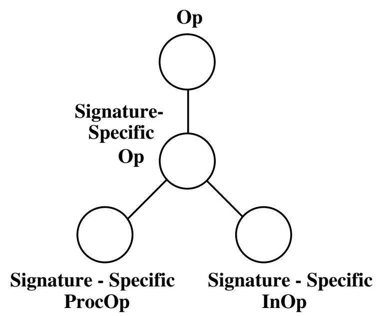

Figure 12: JR operation inheritance hierarchy (from [1]).

VMs.) In particular, the JRVM handles references to virtual machines and support for their dynamic creation, and similar services for remote objects.

4.1.2 Operations and Capabilities

JR provides operations in an object-oriented fashion [1], in which all operations are derived from a base class Op. This base class provides (abstract) methods for invoking and servicing operations. The Op class is specialized according to the signatures of specific operations in a given program. For example, an operation that returns an int and takes a single, String parameter would be represented in Figure 12 as a subclass of Op, whose name would encode the "String to int" signature. Each signature-specific Op class is further specialized into a signature-specific ProcOp class and a signature-specific InOp class. These two classes correspond to the two ways to service operation invocations (Section 2): via a method (an "op-method"; "proc" in SR terminology) or via `inni` statements. These classes provide the concrete methods to support invoking and servicing an operation. For example, the InOp class defines these methods to apply to a queue of invocations declared local to the class. Capabilities for operations also form a hierarchy, with a base class for all capabilities and a subclass for each specific signature used in a given program.

```
public class SimpleOp {
    public static op int myOp1(char);
    public static op int myOp2(char, double);
    public static op int myOp3(double, char);
    public static void main(String [] args) {}
}
```

Figure 13: JR program used to illustrate the classes the translator generates.

4.2 The JR21 Implementation Approach

4.2.1 Generation of signature-specific operation classes

JR21 follows the hierarchy presented in Figure 12 in generating classes for the operations present in a given JR program. For example, consider the simple program in Figure 13. It just declares several operations. (These operations are not serviced anywhere in the given program.) For this simple program, JR21 generates the files listed in Figure 14. Most of these files are: the signature-specific classes, as per Figure 12, for Op (`Op_*`), InOp (`InOp_*`), and ProcOp (`ProcOp_*`); and the corresponding signature-specific classes, mentioned earlier, for capabilities (`Cap_*`). The other files in Figure 14 are for the invocations (`Recv_*`) of these operations. The JR translator generates classes that the program might need rather than actually needs; e.g., it generates both the ProcOp and InOp classes for each operation rather than just one or the other. Also, the JR translator generates classes for some operations it adds to each JR program; e.g., the `*_voidTovoid` classes. The other classes are for the translated version of the actual JR program (`SimpleOp.java`) and for supporting remote objects (`JR*`).

4.2.2 Generic Capabilities

The JR21 scheme for generating signature-specific classes works for generic operations and for most, but not all, generic capabilities. For example, it works for the programs in Figures 5–10, but not for the program in Figure 11.

For the program in Figure 11, the JR21 translator proceeds as follows. It translates the type of `mycap` to a capability that takes a single argument of type `Cap_ObjectToObject` and returns `Cap_ObjectToObject`, i.e., the translator erases the generic type parameter. In the `main` function, `gc` is an instance of type `GenericCap<String>`. In that instance, the member capabil-

```
Cap_charToint.java                     ProcOp_charToint.java
Cap_charXdoubleToint.java              ProcOp_charToint_impl.java
Cap_doubleXcharToint.java              ProcOp_charXdoubleToint.java
Cap_intTovoid.java                     ProcOp_charXdoubleToint_impl.java
Cap_voidTovoid.java                    ProcOp_doubleXcharToint.java
InOp_charToint.java                    ProcOp_doubleXcharToint_impl.java
InOp_charToint_impl.java               ProcOp_intTovoid.java
InOp_charXdoubleToint.java             ProcOp_intTovoid_impl.java
InOp_charXdoubleToint_impl.java        ProcOp_voidTovoid.java
InOp_doubleXcharToint.java             ProcOp_voidTovoid_impl.java
InOp_doubleXcharToint_impl.java        Recv_char.java
InOp_intTovoid.java                    Recv_charToint.java
InOp_intTovoid_impl.java               Recv_charXdouble.java
InOp_voidTovoid.java                   Recv_charXdoubleToint.java
InOp_voidTovoid_impl.java              Recv_double.java
JRSimpleOp.java                        Recv_doubleXchar.java
JRjavadotlangdotObject.java            Recv_doubleXcharToint.java
Op_charToint.java                      Recv_int.java
Op_charXdoubleToint.java               Recv_intTovoid.java
Op_doubleXcharToint.java               Recv_void.java
Op_intTovoid.java                      Recv_voidTovoid.java
Op_voidTovoid.java                     SimpleOp.java
```

Figure 14: Files generated by JR21 for the program in Figure 13.

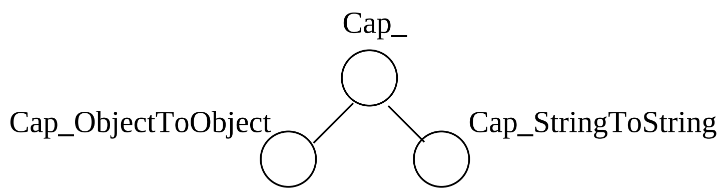

Figure 15: JR 21 capability inheritance hierarchy for the program in Figure 11.

```
JRSimpleOp.java
JRjavadotlangdotObject.java
SimpleOp.java
```

Figure 16: Files generated by JR24 for the program in Figure 13.

ity `mycap` is supposed to take a single argument of type `Cap_StringToString` and to return `Cap_StringToString`. However, `Cap_StringToString` is not a subtype of `Cap_ObjectToObject`. Each of these types is a direct subclass of the base capability class (as described in Section 4.1.2) as illustrated in Figure 15. Therefore, when the Java compiler analyzes this generated code, it complains that the invocation of `gc.mycap` has a type mismatch in both its actual argument and its return value.

4.3 The JR24 Implementation Approach

4.3.1 Generation of signature-specific operation classes

Unlike JR21, JR24 does not generate signature-specific operation classes. Instead, it uses one predefined class for all operations; this class represents the arguments to the operations as a single array of Java Objects. The translator then boxes or unboxes values of primitive types so they can be inserted into or extracted from this array. Similarly, JR24 also uses just one predefined class for capabilities and another for invocations. Since these classes are now unchanged from program to program, they are no longer generated by the translator; instead, they are included in JR's run-time system package.

Thus, for the program in Figure 13, JR24 generates only the files shown in Figure 16. These files have roles similar to the roles of their namesakes in Figure 14 and described in Section 4.2.1

4.3.2 Generic Capabilities

The JR24 scheme solves the problem that JR21 has with some generic capabilities (Section 4.2.2). Because there is just one operation class used to represent operations of all signatures, there is no problem with having subtype classes match as in the JR21 scheme.

5 Performance

We evaluated the performances of JR21 and JR24 with respect to compilation times and execution times on a variety of programs. The overall results show that JR24 is considerably faster for compilation, but there is little difference in the execution times of the code generated by the two versions. The reduction in compilation time is due to JR24 reducing the number of files that the JR translator generates (Section 4): time is saved not only for the time spent actually generating the files (and outputting them), but also in compiling those files. The small differences in execution time are explored later in this section.

Section 5.1 gives a representative collection of benchmarks that demonstrate the overall performance trends. Section 5.2 gives additional benchmarks that reinforce the results given in Section 5.1.

The data presented in this paper were obtained on a 2.8GHz dual-processor system with 1 gigabyte of RAM running Fedora Core 5 Linux (2.6.20-1.2316.fc5smp kernel). The entire set of tests was run multiple times, with insignificant differences between results. Within the set, each individual test was run multiple times. The results reported are the averages of one group of these runs; variances were small. We report elapsed real time, which includes user and system CPU and I/O times; the test system was lightly loaded during the tests, so there was little interference from other tasks running on the system. We also ran these tests on other Linux platforms including a 1.4GHz single-processor system, 2.4GHz single-processor system, and 2.4GHz dual-processor system and on a 2.8GHz dual-processor Windows system. The data obtained on these platforms exhibited trends similar to those we report for our one selected platform, although specific data values, of course, varied. The code generated by JR21 and JR24 was translated and executed using Java 1.5.0_07. That is, the performance differences are due to difference in JR21 and JR24, *not* because the versions are executed using different JVMs. We ran these tests with "adaptive optimizations" in the Java Hotspot byte compiler [10] enabled (the default).

5.1 Representative Benchmarks

We chose a variety of test programs: most we and others had written previously and a few we wrote or adapted specifically for these tests. Some of the programs were from the JR test suite (described later in Section 5.2); some of these are specially designed for assessing execution time performance. Some of the other programs were larger applications (two visualization programs and a program from Reference [2]). Some of these programs are microbenchmarks, while others are macrobenchmarks. Appendix A briefly describes each of these JR benchmarks and gives details on how we ran them.

5.1.1 Compilation Time

Table 1 presents the average compilation times on the representative benchmarks. As the data show, JR24 compilation always requires less time than JR21 compilation: between 67-90%. JR24 outperforms JR21 even on MM-seq, which is sequential JR code involving no user-defined operations or capabilities, because, as noted in Section 4, JR21 produces additional files for all JR programs. JR24 outperforms JR21 by the widest margin on rwVis because that program uses many operations, which requires JR21 to generate and translate considerably more files than JR24 does.

5.1.2 Execution Time

Table 2 presents the average execution times on the representative benchmarks. The data show at most a ±3% difference between the executions of programs run under JR21 and JR24. Although these differences are not significant, we did investigate further a possible cause (parameter passing), as described at the end of this section.

The three programs with execution times marked "n/a" include two visualization programs and a simulation of a distributed file system. Each of these programs requires immediate user interaction, so their execution times are not meaningful.

The differences in the execution times are zero or nearly so for the programs that use virtual machines (vm/basic and vm/many). The execution

Benchmark	Compilation Time		Ratio
	JR21	**JR24**	**JR24/JR21**
MMseq	2.16	1.85	.856
dfsall	4.36	2.93	.672
dpVis	2.57	2.32	.903
rwVis	9.00	6.34	.704
op_charXintXbooleanTodouble	2.49	1.83	.735
op_intXintToint	2.44	1.83	.750
simulationAllPPC	2.52	1.98	.780
simulationAllPPC2	2.51	1.99	.793
timings/asynch	2.24	1.98	.884
timings/ircall	2.30	2.03	.883
timings/irnew	2.31	2.04	.883
timings/locall	2.26	2.00	.885
timings/loop	2.21	1.95	.882
timings/msgcsw	2.28	2.03	.890
timings/pcreate	2.27	2.00	.881
timings/rend	2.28	2.04	.895
timings/semP	2.23	1.95	.874
timings/semV	2.22	1.95	.878
timings/semcsw	2.26	2.04	.903
timings/sems	2.21	1.94	.878
vm/basic	2.57	2.00	.778
vm/many	2.17	1.85	.853

Table 1: Average compilation times (in seconds) and their ratios on representative benchmarks.

Benchmark	Execution Time		Ratio
	JR21	JR24	JR24/JR21
MMseq	16.91	17.12	1.012
dfsall	n/a	n/a	n/a
dpVis	n/a	n/a	n/a
rwVis	n/a	n/a	n/a
op_charXintXbooleanTodouble	9.91	9.86	.995
op_intXintToint	9.86	9.55	.974
simulationAllPPC	19.92	20.66	1.037
simulationAllPPC2	51.98	51.81	.997
timings/asynch	2.94	2.98	1.014
timings/ircall	2.06	2.08	1.010
timings/irnew	2.81	2.89	1.028
timings/locall	2.04	2.08	1.020
timings/loop	1.76	1.82	1.034
timings/msgcsw	2.74	2.80	1.022
timings/pcreate	2.41	2.45	1.017
timings/rend	3.49	3.53	1.011
timings/semP	2.88	2.92	1.014
timings/semV	1.87	1.92	1.027
timings/semcsw	2.67	2.61	.978
timings/sems	2.89	2.88	.997
vm/basic	2.66	2.66	1.000
vm/many	13.69	13.70	1.001

Table 2: Average execution times (in seconds) and their ratios on representative benchmarks.

number of parameters	Execution Time JR21	JR24	Ratio JR24/JR21
0	40.47	41.59	1.028
2	40.36	41.56	1.030
3	41.58	41.39	.995
9	41.21	40.35	.979
15	41.77	41.73	.999

Table 3: Parameter test average execution times (in seconds, for 40,000 iterations with Hotspot optimizations enabled) and their ratios.

times are dominated by the relatively expensive activities of creating virtual machines and of communicating between them, which are the same in JR21 and JR24.

Parameter Passing: A natural concern is that JR24 will take extra execution time due to how it must auto-box (e.g., `int` to `Integer`) and auto-unbox parameters (Section 4.3.1). We ran additional benchmarks to investigate this hypothesis. However, our benchmarks showed little differences in the execution times.

Each of these benchmark programs consisted of a single operation with several integer parameters. The body of the operation just returned the sum of the parameters. The main program invoked the operation many times.[3] We varied the number of parameters and the number of invocations, but execution times using the JR21 and JR24 implementations differed only slightly and in no apparent systematic way. Table 3 shows several typical results we saw. We also ran these tests with "adaptive optimizations" in the Java Hotspot byte compiler [10] disabled (Table 4) with similar outcomes.

We also ran additional benchmarks to investigate the costs of autoboxing/unboxing in Java. While that cost is high relative to the overall cost of a plain Java method invocation, it is not high relative to to the overall cost of a JR invocation. The latter is dominated by the costs of, e.g., object creation for the invocation itself and execution of the code within JR's runtime system for handling an invocation.

[3]These programs are therefore similar to the op_charXintXbooleanTodouble and op_intXintToint programs seen earlier.

number of parameters	Execution Time JR21	JR24	Ratio JR24/JR21
0	66.94	66.85	.999
2	66.29	67.06	1.012
3	66.60	66.85	1.004
9	67.08	67.41	1.005
15	66.74	67.97	1.018

Table 4: Parameter test average execution times (in seconds, for 40,000 iterations with Hotspot optimizations disabled) and their ratios.

5.2 Additional Benchmarks

To gain more data on the relative performances of the JR21 and JR24 implementations, we also ran additional benchmarks. These benchmarks consisted of two JR "vsuites". The first was the main JR implementation vsuite, a suite of roughly 850 test cases for validating (regression testing) the JR implementation [7].[4] The second was the vsuite ("code extract") that accompanies the JR book [2], which is used to ensure that the code fragments in the book actually work; it consists of roughly 250 test cases (available at [7]). Each test case consists of JR source, a test script, and a file with the correct program output for normal programs or a file with correct error messages for erroneous programs. The JR distribution contains a *jrv* tool that launches each test case in the test suite according to the commands in the script file, compares the output against the expected output, and reports the result.

Most of these vsuite tests have short compilation and execution times. Some of these tests consist of a single compilation, but a few executions on different data sets. Table 5 shows the data for running the entire vsuite tests. As shown, JR24 outperforms JR21 overall, with JR24 requiring about 91% of the time that JR21 requires. This number is consistent with the characteristics of the vsuite tests, and the relative compilation and execution performances seen earlier.

[4]The vsuites for the JR21 and JR24 releases differ slightly. The JR24 contains a few additional tests, e.g., for new features in JR24 and for bugs that were formerly present in JR21. A few other tests differ too, for example, due to the wording of error messages. The vsuite actually used for this paper is, therefore, a subset of the JR21 vsuite.

Benchmark	Total Time (mm:ss)		Ratio
	JR21	JR24	JR24/JR21
codeextract2.0	20:36	18:52	.916
vsuite	57:10	52:00	.910

Table 5: Total compilation and execution times and their ratios.

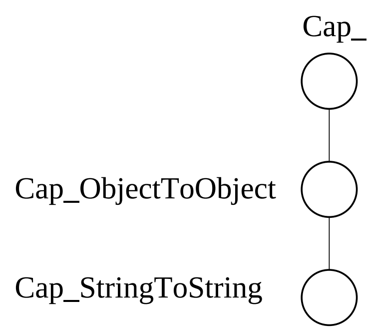

Figure 17: Proper capability inheritance hierarchy for the program in Figure 11.

6 Discussion

Interestingly, we had considered the approach used by JR24 (Section 4.3) for operations and capabilities for our initial implementation of JR1 [1]. However, our experimentation then indicated that, while the approach would reduce compilation times, it would noticeably increase execution times. (We used tests similar to those we used for testing parameter passing reported in Tables 3 and 4 in Section 5.1.2.) Since then, processor characteristics and compiler technology have changed so that this approach no longer has that drawback.

Section 4.2.1 described the JR21 scheme for the generation of signature-specific operation and capability classes. As seen there, JR21 generates several files for each operation. This approach does not work entirely for generic capabilities, as described in Section 4.2.2. A variant of this approach is to generate the same signature-specific operation and capability classes, but to arrange the generated capability classes to form a hierarchy. For example, for the program in Figure 11, the capability classes would be generated to form the hierarchy shown in Figure 17. Doing so would solve the type mismatch

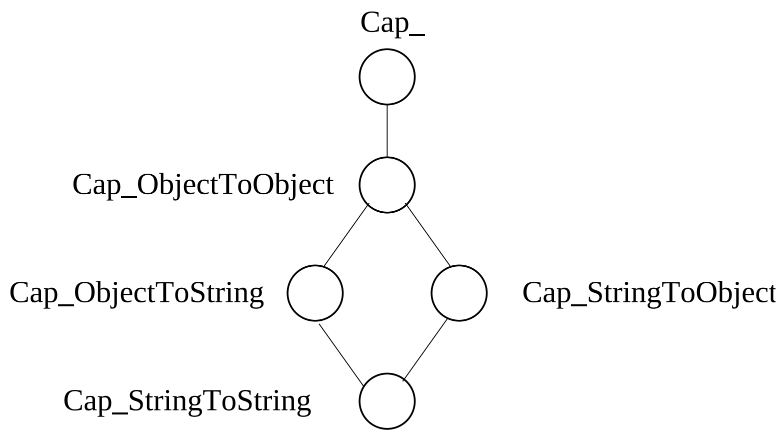

Figure 18: Capability inheritance hierarchy for a variant of the program in Figure 11.

problem described in Section 4.2.2. Note that generating the hierarchy would be feasible within the JR implementation because the JR translator compiles all source files at once. It could thus defer generating the capability hierarchy until it has analyzed the entire program. Although such analysis is possible, it would add some complexity to the implementation and the generated code. For example, a variant of the program in Figure 11 might require other capability classes to be generated and the hierarchy would become more complicated, e.g., as shown in Figure 18. To implement this hierarchy in Java would require the use of interfaces to effect multiple inheritance. Although we did not implement this variant of JR21, we believe its performance would be roughly the same as JR21's performance. Its performance would likely be just slightly slower during translation time, for the reasons mentioned previously, and would be about the same during execution time.

In a more general context, numerous other languages have extended Java. Many of these extensions have been for concurrency. Some earlier efforts include JCilk [11], Ajents [12], JavaParty [13], ARMI [14], Java/DSM [15], Charlotte [16], and Communicating Java Threads [17]. More recent efforts include JAC [18], simpA [19], and Coqa [20]. The earlier efforts predate generics in Java and the reports on the more recent efforts do not address whether or how they deal with generics.

7 Conclusion

This paper described the new JR2 language features of generic operations and generic capabilities, motivated by the changes in the underlying Java language. It then described our initial (JR21) implementation of generic operations and capabilities, and the problem with that approach. The paper then described the approach the JR24 implementation uses to fully implement generic operations and capabilities. It described the benchmark comparisons we used to assess the compilation and execution time performances of JR21 and JR24. The JR24 implementation reduces compilation times, mainly due to reducing the number of files generated during JR program translation, without noticeably impacting execution times.

Acknowledgements

Others in the JR group — Billy Yan-Kit Man, Erik Staab, and Ingwar Wirjawan — assisted with this work. The anonymous reviewers gave constructive comments that led to improvements in this paper.

A Description of JR Benchmarks

Tables 6 and 7 present brief descriptions of the JR benchmarks that we used in Section 5.1. As noted in that section, most of these benchmarks come from the JR main vsuite or the JR book vsuite (both available at [7]).

Some of the benchmarks require command-line or hard-coded parameters. We used the following in running the benchmarks:

- MMseq: 1,000,000 multiplications of 2 10×10 matrices.
- op_charXintXbooleanTodouble and op_intXintToint: 2,000 sends and receives.
- all timings tests: "10 10 1", each trial consists of 10×10 iterations, run as 1 trial.
- simulationPPC and simulationPPC2: 1,000 voters.
- vm/many: 20 total virtual machines, maximum of 3 at a time.

Benchmarks	Approx. # of lines	Description	Language features used
MMseq	64	Matrix multiplication	sequential code
dfsall	644	Distributed file system	vm, remote object, inni, send, receive, reply, forward, capability
dpVis	2178	Visualization of the dining philosophers problem	process, send, receive, sem, P, V, operations, capability, vm, remote object
rwVis	2678	Visualization of the readers/writers problem	vm, remote object, inni, send, receive, reply, forward, capability
op_charXintXboolean-Todouble	17	sends and receives on an operation that takes a char, an int, and a boolean and returns double	send, receive
op_intXintToint	17	sends and receives on an operation that takes two int and returns int	send, receive
simulationAllPPC	41	Election simulation	process, inni, send, receive, operation
simulationAllPPC2	45	Election simulation using an array of operations	process, inni, send, receive, array of operation

Table 6: Brief descriptions of JR benchmarks (part 1 of 2).

Benchmarks	Approx. # of lines	Description	Language features used
timings/asynch	60	Asynchronous send/receive	call, operation, inni, send, receive
timings/ircall	68	Interclass call, no new process	call, operation, remote
timings/irnew	75	Interclass call with new process creation	call, operation, remote
timings/locall	59	Local call	call
timings/loop	57	Overhead of an empty loop	
timings/msgcsw	81	Message passing requiring context switch	send, receive, P, V, sem
timings/pcreate	64	Process create	P, V, sem, send
timings/rend	68	Rendezvous	send, call, inni, operation
timings/semP	60	Semaphore P operation	P, V, sem
timings/semV	57	Semaphore V operation	P, V, sem
timings/semcsw	79	Semaphore requiring context switch	send, P, V, sem
timings/sems	57	Semaphore operations P and V	P, V, sem
vm/basic	38	Remote objects on a JR virtual machine	remote object, vm, quiescence operation
vm/many	70	many virtual machines	remote object, vm array

Table 7: Brief descriptions of JR benchmarks (part 2 of 2).

References

[1] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 578–608, May 2004.

[2] Ronald A. Olsson and Aaron W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer International Series in Engineering and Computer Science; SECS 774. Boston : Kluwer Academic, 2004.

[3] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language : Concurrency in Practice*. Benjamin/Cummings Pub. Co., 1993. `http://www.cs.arizona.edu/sr/`.

[4] *Sun Developer Network*, 1994-2005. `http://java.sun.com/`.

[5] *JDK 5.0 Documentation*, 2004. `http://java.sun.com/j2se/1.5.0/docs/index.html`.

[6] Hiu Ning (Angela) Chan. Enhancing the JR concurrent programming language with new Java 5.0 features. Master's thesis, University of California, Davis, Department of Computer Science, December 2005. `http://www.cs.ucdavis.edu/~olsson/students/`.

[7] JR distribution. `http://www.cs.ucdavis.edu/~olsson/research/jr/`.

[8] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[9] G.R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[10] *Advanced Programming for the Java 2 Platform*, 2007. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html`.

[11] *JCilk — A Java-Based Multithreaded Programming Language.* `http://publications.csail.mit.edu/abstracts/abstracts05/jsd_angelee_cel/jsd_angelee_cel.html`.

[12] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile Java applications. In *ACM 1999 Java Grande Conference*, pages 15–24, 1999.

[13] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[14] R. Raje, J. Williams, and M. Boyles. An asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[15] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.

[16] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[17] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *WoTUG 20*, pages 48–76, 1997.

[18] Max Haustein and Klaus-Peter Löhr. JAC: declarative Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[19] Alessandro Ricci and Mirko Viroli. simpA: A simple agent-oriented Java extension for developing concurrent applications. In *Proceedings of Languages, Methodologies and Development Tools for Multi-agent Systems (LADS'007)*, 2007. `http://lia.deis.unibo.it/confs/lads/papers/4.3%20paper_37%20(ricci).pdf`.

[20] Yu David Liu, Xiaoqi Lu, and Scott Smith. Coqa: Concurrent objects with quantized atomicity. In *Proceedings of the 18th International Conference on Compiler Construction*, 2008. to appear.