**Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study**

by

AARON WILLIAM KEEN

B.S. (University of California, Davis) 1997
M.S. (University of California, Davis) 2000

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Ronald A. Olsson (Chair)

Professor Premkumar Devanbu

Professor Raju Pandey

Committee in Charge

2002

**Integrating Concurrency Constructs with Object-Oriented Programming Languages: A Case Study**

## Abstract

Concurrent programming carries the promise of increased program performance from overlapping the execution of cooperating components. Explicitly concurrent programming languages provide constructs and abstractions to assist in the design and implementation of concurrent programs. Such languages provide support for creating threads (separate flows of execution), sending and receiving messages, and coordinating cooperation through synchronization. Furthermore, many languages provide abstractions that blur the lines between physical machines and that mask communication between physical machines.

Object-oriented programming languages also provide constructs and abstractions that can aid in program design. These languages provide mechanisms to create abstract types that encapsulate data and behavior. Using such mechanisms, programmers can create object libraries that can be reused to increase productivity and decrease errors. Many object-oriented programming languages provide such object libraries.

This dissertation work integrates an object-oriented language with a flexible concurrency model to provide a well-defined concurrent object-oriented language in which parallel and distributed programs and libraries can be written. The specific work presented integrates the SR concurrency model with the Java programming language. The result is the design and implementation of the JR concurrent object-oriented programming language. JR supports a rich concurrency model for writing tightly-coupled concurrent programs and for writing reusable concurrent libraries. Moreover, JR provides support for writing programs with dynamic communication paths.

This dissertation presents, first, the design and implementation of the JR programming language. Next, an extension to Java's exception handling model is discussed. This extended exception handling model completes the integration of the SR concurrency model with the Java programming language. Specifically, Java's exception handling model is extended to support exception handling during asynchronous invocation. Finally, the dissertation presents a study in the design and analysis of several inter-operation invocation selection mechanisms. The proposed mechanisms provide, unlike preexisting mechanisms, access to the entire set of pending messages when determining which message to receive.

To Diana.

# Contents

# List of Figures

# List of Tables

## Acknowledgements

This work would not have been possible without the support, guidance, and encouragement of my advisor, Dr. Ron. He supported my return to UC Davis and helped to make the transition easy. His dedication to his students and his teaching have inspired my pursuit of an academic career. Thank you, Ron.

I have also been very fortunate to have had two mentors that have contributed greatly to my academic development. I thank Raju Pandey and Prem Devanbu for each adopting me, in some sense, as one of their own students. The seminars, meetings, and discussions will be sorely missed.

Graduate life would certainly have been less interesting if not for the group of cohorts that make up the software systems contingent at UC Davis. I thank Michael Haungs, Eric Wohlstadter, Stoney Jackson, Earl Barr, and Eric Hyatt for the many interesting and entertaining discussions about research, the field of computer science, computer games, and any other random topic that happened arise. Keep up the good work. And remember, software systems research is *not* irrelevant!

I must thank Kim Reinking for all that she has done and continues to do for the graduate students in the department. Kim has been a great resource and friend. Thank you for championing the causes of the graduate students and for making my return to Davis as simple as it was. I thank Mary Brown, Debbie Chadwick, Melinda Day, Michele Fulton, Patty Graves, Meshell Hays, Maggie Rimoldi, and Josie Valdez for keeping the department running on a day-to-day basis. I also thank Ken Jones, Ken Gribble, Jim Moersfelder, Babak Moghadam, Lisa Thurston, Terri Knight, and Barb Weston for all of their help and advice on restoring, repairing, and securing the research machines.

My family has provided continued support for my education. Thanks Mom, Grandma, Grandpa, Greg, Cindy, Mike, Ashley, Zach, and Jeremy.

Finally, I thank Diana for the encouragement to apply to graduate school and the support and inspiration to continue. I love you.

# Chapter 1

# Introduction

Concurrent programming carries the promise of increased program performance from overlapping the execution of cooperating components. The simplest concurrent programs may actually be sequential programs that cooperate using some form of interprocess communication. On UNIX, for example, two programs communicating through a pipe may be considered, as a whole, a concurrent program. A program written as separate parts that communicate using sockets may also be considered a concurrent program. Though such "programs" are, arguably, concurrent programs, writing concurrent programs in such a manner is complicated by the lack of abstractions provided. This work focuses on language-level support for writing tightly-coupled, i.e., closely cooperating, concurrent programs.

Explicitly concurrent programming languages provide constructs and abstractions to assist in the design and implementation of concurrent programs. Such languages provide support for creating threads (separate flows of execution), sending and receiving messages, and coordinating cooperation through synchronization. Furthermore, many languages provide abstractions that blur the lines between physical machines and that mask communication between physical machines. At the opposite extreme, implicitly parallel languages rely upon compiler and run-time support for splitting a program into interacting components and for defining synchronization between the components.

Object-oriented programming languages also provide constructs and abstractions that can aid in program design. These languages provide mechanisms to create abstract types that encapsulate data and behavior. Using such mechanisms, programmers can create object libraries that can be reused to increase productivity and decrease errors. Many object-oriented programming languages provide such object libraries.

Specifically, this work integrates an object-oriented language with a flexible concurrency model to provide a well-defined concurrent object-oriented language in which parallel and distributed programs and libraries can be written. The specific work presented integrates the SR [4, 3] concurrency model with the Java [38] programming language. The result is the design and implementation of the JR concurrent object-oriented programming language. The rest of this section outlines key aspects of this work, summarizes the contributions, and outlines the remaining chapters.

## 1.1 The JR Programming Language

Java has proven to be a relatively clean and simple 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). Though these features are useful, they offer little flexibility in the design and implementation of concurrent programs.

The JR programming language is an extension of Java with support for the concurrency constructs that define SR's concurrency model. JR, as a language, extends Java's syntax in a manner that aims to retain the "feel" of Java. JR adds to or expands on the structural, concurrency, and communication abstractions provided by Java. JR provides structural abstractions for remote virtual machines and remote objects; concurrency abstractions for dynamic thread creation, dynamic virtual machine creation, and dynamic remote object creation; and communication abstractions for remote method invocation, asynchronous method invocation, and multi-way receive. These abstractions provide the programmer with increased flexibility for designing and implementing concurrent programs. In particular, JR allows programmers to write tightly-coupled concurrent programs with dynamic communication paths.

## 1.2 Exception Handling during Asynchronous Invocation

Asynchronous method invocation facilitates the dynamic creation of concurrently executing threads and the communication between such threads. When a method is asynchronously invoked, the invoking thread continues to execute while another thread executes the body of the invoked method. Such concurrent execution can benefit both the design of and the performance of a program. The use of asynchronous method invocation, however, complicates the use of exceptions and, in particular, the handling of thrown exceptions.

The exception handling mechanisms provided by sequential programming languages rely upon the call stack for the propagation of exceptions. A thrown exception is propagated (either implicitly or explicitly) up the call stack until an appropriate handler is found. Unfortunately, the reliance of such mechanisms on the call stack is inadequate for handling exceptions thrown from an asynchronously invoked method because the invoking method may no longer be executing. This work addresses this problem by requiring the specification of handlers (in the form of objects) for exceptions thrown from methods invoked asynchronously. A handler is passed as an implicit argument to the asynchronously invoked method and, as such, the handler resides at the site of the executing method.

## 1.3 Inter-operation Invocation Selection Mechanism

Many concurrent programming languages support message passing as a form of thread synchronization. Invocations (messages) are sent to and received from an operation (message queue). An invocation is generated when a process invokes an operation, optionally passing it parameters. Once generated, the invocation is added to the set of pending invocations associated with the invoked operation. An invocation is removed from an operation's set by a servicing process using an invocation servicing mechanism. Such a mechanism, depending on the language, may allow selection of an invocation from a single operation (a receive) or from one of potentially many operations (a multi-way receive).

This work addresses the short-comings of selection mechanisms for multi-way receives provided by other languages. Specifically, this work provides a new inter-operation invocation selection mechanism that provides simultaneous access to all pending invocations during selection.

## 1.4 Contributions

The contributions of this work are the design and implementation of the JR programming language, the study and implementation of support for exception handling during asynchronous invocation, and the design, analysis, and implementation of different inter-operation invocation selection mechanisms.

### 1.4.1 The JR Programming Language

It is expected that JR will be useful as a research and a teaching tool. The design of the JR programming language should be beneficial to the future design and implementation of other concurrent object-oriented programming languages. Furthermore, JR is useful as a language for implementing concurrent programs.

JR has been used as a research tool in the exploration of exception handling during asynchronous invocation and in the study of different inter-operation invocation selection mechanisms. In addition, JR has been used in both undergraduate and graduate courses studying concurrent programming.

### 1.4.2 Exception Handling during Asynchronous Invocation

The exception handling model explored by this work should be useful in other concurrent languages and systems (e.g., CORBA [62]) that support exceptions and asynchronous invocation.

### 1.4.3 Inter-operation Invocation Selection Mechanism

Languages that support multi-way receives can benefit from the selection mechanisms analyzed in this work. In fact, since the resulting solution does not depend on the object-oriented features of JR, support for the solution can be provided by concurrent languages that are not object-oriented (e.g., SR).

## 1.5 Organization

The remainder of this dissertation is organized as follows. Chapters 2, 3, and 4 present background material in support of the discussion of the JR programming language. Chapter 2 provides background material on concurrent programming models, synchronization primitives, object-oriented programming languages, and concurrent object-oriented programming languages. Chapter 3 presents an overview of the concurrency support provided by Java and used in this work. Chapter 4 describes the SR programming language and its concurrency model.

Chapters 5, 6, and 7 present the contributions of this work. Chapter 5 presents the JR programming language and its implementation. Chapter 6 discusses the final piece of the integration of SR's concurrency model with the Java programming language: the handling of exceptions during asynchronous method invocation. Chapter 7 presents the design and analysis of four inter-operation invocation selection mechanisms.

Finally, Chapter 8 reflects on and concludes the dissertation.

# Chapter 2

# Background

This chapter provides brief background information on object-oriented programming, concurrent programming models, concurrent programming constructs, concurrent object-oriented languages, and the inheritance anomaly.

## 2.1 Object-Oriented Languages

The goals, concepts, and paradigms of object-oriented programming are discussed in [84] and [26]. Object-oriented programming languages facilitate the encapsulation of a set of data and the operations associated with a set of data into abstract types. This encapsulation provides a powerful means for modular programming and a great potential for module reuse. The following discussion focuses on class-based object-oriented languages.

An *object* is an encapsulation of state and operations that apply to that state. A *class* is a template from which objects are created. Just as a blueprint defines the features of a building, a class defines the features of the objects created from it. An object is created by instantiating a class and is, therefore, commonly referred to as an *instance* of its defining class. The state of an object is represented by a set of instance variables. The operations that may modify the state of a particular object are called instance *methods*. A class may also define class variables and class methods, which define state and behavior at the class-level.

A class definition can be viewed as consisting of a specification and an implementation. The specification of a class defines the external interface "exported" by instances of that class. An instance of a class is used by invoking instance methods defined in the specification of the class. The implementation of a class defines the actions taken when a method in the specification is invoked (i.e., the implementation defines the statements that are executed upon invocation).

True object-oriented languages support inheritance and dynamic method dispatch. In such a language, a new class may be derived from an existing class. The newly defined class is called a `subclass` of the existing class, which is referred to as the parent class or `superclass`. A subclass inherits the specification and, by default, the implementation of its superclass. A subclass may extend the inherited specification by defining new methods. A subclass may also modify its inherited implementation by redefining the inherited methods.

An instance of a subclass can be used as an instance of its parent class because it supports the specification defined in its parent class. This allows a programmer to use instances of a subclass in place of its parent class. Moreover, when a method is invoked on an instance, the class that defines the instance is used to determine which definition of the method is actually executed. Therefore, if a subclass redefines an inherited method, all invocations of that method through instances of the subclass will invoke the redefined method rather than the inherited method. Dynamic method dispatch ensures that the behavior redefined by a subclass is maintained even when an instance of the subclass is used as an instance of its superclass. For example, Figure 2.1 demonstrates the redefinition of a print method and the invocation of that print through dynamic dispatch.

```
public class A
{
  public void print()
  {
    System.out.println("Class A's print!");
  }
}

public class B extends A
{
  public void print()
  {
    System.out.println("Class B's print!");
  }
}
```

```
public class Main
{
  public static void main(String [] args)
  {
    A a = new B(); // create a B object
    a.print(); // prints "Class B's print!"
  }
}
```

Figure 2.1: Simple dynamic dispatch example.

Classes can be viewed as defining a behavior that instances share. Likewise, inheritance is used by subclasses to share all or part of the behavior defined in an existing class. Under inheritance, each object is an instance of a specific class. The association between objects and classes does not, typically, change at run-time.

## 2.2 Concurrent Programming Models

The authors of a recent survey [76] define six different concurrent programming models and analyze these programming models according to six criteria to assess their suitability for realistic portable concurrent programming. The six concurrent programming models, in order of decreasing abstraction, are:

- Nothing Explicit, Parallelism Implicit

  All issues of parallelism are abstracted away so that the programmer need not be concerned with expressing or coordinating parallel computation. All identification, decomposition, and coordination of parallelism is performed by the language support system (i.e., compiler and/or run-time system).

- Parallelism Explicit, Decomposition Implicit

  Programmers need only express the potential for parallelism in a particular program. The actual decomposition of the program into separate threads and the coordination of these threads is done implicitly by the language support system.

- Decomposition Explicit, Mapping Implicit

  The programmer must make all decisions about how the computation is broken into units. The language support system maps the parallel units onto the available processors and coordinates their execution.

- Mapping Explicit, Communication Implicit

  Not only must the programmer break the computation into parallel units, but the programmer must also map those units to specific processors for execution. The language support system provides for the communication and synchronization between the parallel units.

- Communication Explicit, Synchronization Implicit

  In addition to identifying, decomposing, and mapping parallel units, the programmer must also explicitly implement the communication between the units. The language support system only deals with synchronization issues.

- Everything Explicit

  Though a specific language in this model may provide abstractions for the concurrent mechanisms, the model abstracts nothing from the programmer. The programmer must explicitly identify, decompose, and map parallel units to different processors as well as implement the communication and synchronization between different units.

These concurrency models are analyzed according to the following criteria: ease to program, software development methodology, architecture-independence, ease of understanding, guaranteed performance over a range of architectures, and cost measures of the programming models operations. Of these criteria, the first four relate to the model as a target for software development, whereas the last two address the requirement for execution on real parallel machines. The authors recognize that the most abstract (high-level) models satisfy the first four criteria but cannot (generally) provide predictable performance over a variety of architectures. Conversely, the survey authors believe that work on the more explicit (low-level) models should be abandoned in favor of abstraction, even though the explicit models do provide performance guarantees. As such, the survey authors believe that research on concurrent languages that fall into the mid-range models will yield sufficient abstractions while guaranteeing a certain degree of performance.

The JR programming language falls, most apparently, into the "Everything Explicit" category. Despite what the authors of the survey suggest, this does not mean that JR should be abandoned. Though JR does require the programmer to explicitly identify, decompose, and map units to different processors and to implement communication and synchronization between units, it provides many abstractions for doing so. The operation abstraction, for instance, leverages the programmer's familiarity with method invocation and generalizes it to support message passing, thread creation, and rendezvous synchronization. Moreover, explicit control is beneficial in writing tightly-coupled concurrent programs that support load-balancing, co-location, and heterogeneous physical machines.

## 2.3 Concurrent Programming

When writing concurrent programs in one of the explicit models discussed in Section 2.2, a programmer may be required to express the parallelism that may be exploited, map parallel units onto processors, and coordinate the interaction of these parallel units. Three different issues that distinguish distributed programming from sequential programming are identified in [10]. These issues are the use of multiple processors, the cooperation among processors, and the potential for partial failures. The remainder of this section addresses how these issues are realized in programming languages.

### 2.3.1 Expressing Parallelism

Both [5] and [10] discuss different mechanisms to express parallelism. This section discusses only the explicit expression of parallelism. Furthermore, only language-level parallelism is discussed; parallelism supported through the use of separate operating system processes and inter-process communication is not discussed.

#### 2.3.1.1 Threads

The most common expression of parallelism is the thread. Threads represent separate flows of control that typically share an address space and may execute concurrently. A thread is often declared in a manner similar to a procedure or created through an asynchronous invocation of a procedure. Once created, a thread executes the sequential code associated with its declaration. True concurrency requires multiple processors to execute different threads at the same time. Even if run on a single processor, it is often useful to design a program as a set of concurrently executing threads.

#### 2.3.1.2 Coroutines

Coroutines are similar to procedures but support a modified mechanism to transfer control between different coroutines. A coroutine may transfer control to another coroutine using a `resume` statement. Executing a resume statement causes the current coroutine to store enough state to later return to the instruction following the resume statement. Control is then transferred to the invoked coroutine, which begins execution at the instruction stored after its last resume (or at the beginning if it had never been resumed before).

Coroutines can be used to simulate the execution of multiple processes, but cannot express true parallelism since only a single coroutine may execute at a time.

#### 2.3.1.3 Cobegin

The `cobegin` statement can be used to concurrently execute a set of statements. A `cobegin` statement terminates when each statement in the set of concurrently executing statements has terminated. A similar mechanism is a parallel loop which executes each iteration of the loop in parallel.

#### 2.3.1.4 Objects

Though concurrent object-oriented languages are discussed further in Section 2.4, [10] lists four ways to express parallelism by extending the sequential object model.

- Allow an object to execute without receiving a message (i.e., associate a process with the object to create an active object).
- Allow an object to continue executing after replying to a message.
- Allow messages to be sent to multiple objects at the same time.
- Allow the message sender to continue executing after sending a message.

### 2.3.2 Mapping onto Processors

As mentioned in Section 2.2, parallel units may be implicitly or explicitly mapped to processes within a system. Three approaches to mapping parallel units to processors are discussed in [10]. These approaches are independent of whether or not the mapping is explicit or implicit. A processor mapping can be fixed at compile time, at run-time, or changed dynamically. Compile time mapping provides little flexibility, but allows the programmer to rely upon the location of parallel units. Run-time mapping assigns a parallel unit to a processor upon creation. Such an approach can be used to facilitate limited load-balancing. Under dynamic assignment, a parallel unit can potentially migrate between processors. If parallel units are capable of communicating, then a system that supports migration must also support mechanisms to locate migrating units or provide location-transparent communication channels.

### 2.3.3 Synchronization Constructs: Shared Variables

Several synchronization constructs use shared variables [5]. The first and most primitive construct is busy-waiting. A process that repeatedly tests a shared variable until its value satisfies a condition is said to be busy-waiting. Busy-waiting can be used to implement mutual exclusion, as demonstrated in [68] and [75]. However, using busy-waiting to implement synchronization is error-prone and inefficient.

Semaphores provide a simple synchronization abstraction with a well-known interface. A **P** operation delays until the count associated with a semaphore is greater than zero and then decrements the count. A **V** operation increments the count associated with a semaphore. Semaphores can be used to provide mutual exclusion by simply preceding each critical section with a **P** operation and ending each critical section with a **V** operation. Condition synchronization (e.g., waiting for a boolean condition) can be implemented using semaphores and shared variables to represent each condition.

Though not quite as low-level as busy-waiting, semaphores can also be error prone since they are unstructured. A conditional critical region (CCR) provides a structured notation for specifying synchronization of critical sections. Shared variables are defined as part of the CCR and access to the shared variables is only permitted in CCR statements. Each CCR statement blocks the executing process until the statement guard is true. Mutual exclusion is ensured within the CCR because only a single process at a time can execute within a particular CCR.

Though more structured than semaphores, synchronization using conditional critical regions still requires the synchronization code to be scattered throughout a program. Monitors provide a centralized abstraction to implement synchronization. A monitor defines a set of operations and a set of shared variables, which may only be accessed through the operations. Only a single process may be executing within a monitor (more specifically, within an operation of a monitor). Condition synchronization is implemented using shared condition variables within a monitor. A process executing within a monitor may wait on a condition variable, which causes that process to relinquish exclusive access to the monitor and block on the condition variable. Likewise, a process executing within a monitor may signal a condition variable, which causes a process waiting on that condition variable to awaken.

A more generalized mechanism to provide synchronization over the operations within a module are path expressions. Path expressions are used to define the permissible interleavings of operation invocations. For example, consider a statement such as the following (taken from [5]):

path N : (1 : (deposit); 1: (fetch)) end

It ensures that activations of deposit are mutually exclusive, activations of fetch are mutually exclusive, each fetch is preceded by a deposit, and the number of completed deposit invocations is never more than `N` greater than the number of completed fetch invocations. The synchronization code for path expressions is generated by the compiler. Similar high-level synchronization constructs are discussed in Section 2.5.

Additional high-level synchronization mechanisms rely on transitions between abstract states (see Section 2.4 for an example). Using such mechanisms, the programmer defines a set of states that apply to the operations defined within a module or object. Each state defines the set of operations that may be invoked when the module is in that state. When a method is invoked, the current state is checked to determine if such an invocation is permissible. If the invocation may proceed, then the method is executed and the state is, potentially, changed. If the invocation is is not permissible in the current state, then the invocation is deferred until the module enters a state in which the invocation is valid.

### 2.3.4 Synchronization Constructs: Message Passing

Several synchronization constructs use message passing [5, 10]. In its most primitive form, message passing allows data to be sent from one process to another through a communication channel. The sending process executes a `send` statement that specifies the communication channel and the data to be sent. A receiving process executes a `receive` statement that specifies the communication channel from which to receive a message and the variables in which to store the received data. A communication channel can be specified through *direct naming* in which the participating processes are named directly. A more sophisticated scheme allows a communication channel to be defined using global names. Under such a scheme, messages are sent to specified global *mailboxes* which hold the messages until another process receives from the mailbox.

Message passing may be either synchronous or asynchronous. In synchronous message passing, the sending process and the receiving process will each block until the other has also initiated communication. In asynchronous message passing, the sending process does not block on invocation of the `send` statement. However, even in asynchronous message passing, the receiving process will typically block until the message has arrived.

Higher-level message passing constructs include remote procedure call and rendezvous. A remote procedure call attempts to provide the semantics of a standard procedure call by hiding the message passing implementation. The invoker of the remote procedure must block waiting for the procedure to return. At the destination, a process may be created to execute the body of the procedure and return the result or a single process may service each invocation of the procedure.

A similar higher-level message passing construct is rendezvous. An invocation of the remote "procedure" is actually serviced by a statement that accepts invocations, executes a block of statements, and, optionally, returns a result. Such a statement causes the executing process to block until an invocation arrives at which point the blocked process actually executes the rendezvous code instead of creating a new process. Rendezvous is similar to synchronous message passing, but differs in that the "sender" remains blocked until the "receiver" has completed executing the rendezvous statement's code block.

### 2.3.5 Partial Failures

Distributed programs may suffer from partial failures and, therefore, need mechanisms to account for them [10]. A distributed program consists of a set of processors that cooperate to solve a problem. Unlike in a sequential program, a processor failure need not necessarily result in the termination of the entire program. A distributed program may be explicitly designed to tolerate such failures. If the program detects that a processor has crashed, then it may be possible to reassign the work for which the failed processor was responsible.

Support for fault tolerance may be provided by the run-time system of a programming language. A run-time system may implement a reliable remote procedure call that ensures successful execution of the procedure even when there are communication failures. However, such support must take care to avoid violating the procedure call semantics of the programming language. If a remote procedure assumes *exactly-once* semantics, then duplicate invocations due to communication failure may result in erroneous behavior. Many languages and systems that support distributed programming define procedure invocation semantics that account for communication errors. In such a language, the programmer must ensure that procedures invoked remotely adhere to the semantics defined by the language. Alternate invocation semantics include *at-least-once* and *at-most-once*.

A language may also support grouping a set of operations together into an *atomic transaction*. An atomic transaction is indivisible. If any operation within the atomic transaction fails, then the entire transaction fails. An atomic transaction will only commit when the entire set of operations defining the transaction have successfully completed.

## 2.4 Concurrent Object-Oriented Languages

A recent survey [18] classifies the different approaches in object-oriented concurrent programming languages into three "coarse" categories. The three categories are: the *library* approach, the *integrative* approach, and the *reflective* approach. These categories reflect the manner in which concurrency components are provided by the object-oriented language.

### 2.4.1 Library Approach

The *library* approach uses object-oriented concepts (i.e., encapsulation and abstraction) to provide libraries that encapsulate concurrency components. For example, in Java, a thread is represented by the library class Thread. A programmer can extend the Thread class and define the thread's activity by overriding the `run` method as demonstrated in Figure 2.2. A library approach is oriented toward system builders, providing a set of abstractions from which concurrent systems can be constructed.

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

Figure 2.2: A user-defined thread in Java.

### 2.4.2 Integrative Approach

The *integrative* approach unifies concurrency concepts with object-oriented concepts. The survey defines three dimensions of integration: active objects, synchronized objects, and distributed objects. An active object derives from the integration of the concept of an object and the concept of a process (thread). Active objects are further categorized by the amount of intraobject concurrency, reactivity versus autonomy, acceptance of messages, and the concept of a body. The amount of intraobject concurrency can range from serial active objects, in which only a single invocation is computed at a time, to fully concurrent active objects, in which concurrency within the object is not restricted. Between these two extremes, the amount of concurrency is restricted by either the language or the programmer. A reactive object only reacts to events (method invocations), whereas an autonomous object can compute in isolation. Messages (or invocations) can be either implicitly or explicitly accepted. Many autonomous active objects rely on the concept of a body: a centralized operation that explicitly describes the behavior of the object. Figure 2.3, taken from [18], demonstrates an active queue in POOL. A Queue object has a single thread of control that repeatedly accepts messages by executing the statements in the body (between the `BODY` and `YDOB` keywords).

Synchronized objects integrate the concept of synchronization with the concept of an object. Such an integration can allow for the specification of intraobject synchronization, behavioral synchronization, and interobject synchronization. Intraobject synchronization ensures the integrity of an object state by enforcing exclusion between operations (e.g., readers and writers cannot access a file simultaneously). Behavioral synchronization modifies the behavior of an object to delay some invocations until a valid condition is met (e.g., a bounded buffer will delay insertions if the buffer is full). Interobject synchronization ensures consistency between a number of communicating objects, such as an atomic transaction. Synchronization schemes that address these different levels of concurrency control include (see Section 2.3.3 for discussion): path expressions, which specify the possible interleaving of invocations; behavior replacement, which allows an object to modify its behavior based on the invocations that it has serviced; abstract states, which represent a set of enabled methods; guards, which associate a boolean condition with each method; and annotations, which are used to declare compatibilities among methods. Figure 2.4, taken from [55], demonstrates the integration of synchronization based on abstract states to implement a

```
CLASS Queue
   ... // define local variables
   METHOD enq(item: T)
   BEGIN
     cell!put(rear,item);
     rear := (rear + 1) MOD size
   END enq

   METHOD deq(): T
   BEGIN
     RESULT cell!get(front);
     front := (front + 1) MOD size
   END deq

   BODY
     DO
       IF empty() THEN ANSWER(enq)
       ELSIF full() THEN ANSWER(deq)
       ELSE ANSWER ANY
       FI
     OD
   YDOB
END Queue
```

Figure 2.3: An active queue in POOL.

bounded buffer in ACT++. After each invocation completes, the next abstract state is computed to determine the current set of valid invocations. An invocation not present in the current abstract state will be delayed.

Distributed objects treat objects as the unit of distribution and replication. As the unit of distribution, objects can be placed in physically distributed memory. Method invocation is mapped onto a message passing mechanism to transparently support such distribution. Such distribution can increase the amount of true parallelism since the objects need not share a single processor. Moreover, with the state of each object sufficiently encapsulated, objects may be migrated between processors to improve accessibility and load-balancing. Replication of distributed objects can increase accessibility, but may require support to maintain object consistency.

Though the integrative approach elegantly unifies concurrency concepts with object-oriented concepts, the approach typically suffers an inheritance anomaly, which will be discussed in Section 2.5. Additionally, the distribution of objects can suffer from commu-

```
Class b-buf : ACTOR
{
      int in, out, buf[SIZE];
   behavior:
      empty = put;
      partial = put, get;
      full = get;
   public:
      void b-buf()
      {
         in = out = 0;
         become empty;
      }
      void put()
      {
         in++; // store an item
         if (in == (out + size)) become full;
         else become partial;
      }
      void get()
      {
         out++; // remove an item
         if (in == out) become empty;
         else become partial;
      }
   }
}
```

Figure 2.4: A bounded buffer in ACT++.

nication failures, modified invocation semantics (i.e., not exactly-once), object duplication, and partitioning of `class` variables into multiple copies.

### 2.4.3 Reflective Approach

The *reflective* approach uses reflection mechanisms to provide concurrency components at the meta-level. The meta-components of a reflective architecture can be modified to specialize message sending, receiving, buffering, selection, method lookup, execution, and state accessing. For example, by specializing the message-buffering component to store messages in a queue and specializing the execution component to associate a thread with the object, an active object can be realized. Reflective architectures are still in their infancy, but will likely provide powerful mechanisms to allow dynamic modification of the functionality of a system.

For example, in Actalk [20], which is based on SmallTalk, a class may extend the `Activity` class, which describes the internal activity of an active object, to provide various synchronization schemes. Figure 2.5, taken from [20], demonstrates the extension of the `Activity` class to provide synchronization through abstract states. The `initialize`, `nextMessage`, and `eventComplete` methods are redefined to support transitions between abstract states and message selection based on the current state. The `BufferEnabledSetsActivity` class uses the extended `Activity` class to provide synchronization for a bounded buffer. The example does not show the actual implementation of the bounded buffer.

## 2.5 Inheritance Anomaly

In concurrent object-oriented languages, a conflict exists between inheritance and synchronization constraints that often requires redefinition of inherited methods in order to maintain the integrity of concurrent objects. This conflict has been deemed the *inheritance anomaly*. Synchronization constraints are used to determine if a method can be successfully invoked. If the constraints are not satisfied, then the invocation is deferred. For example, Figure 2.6, taken from [55], demonstrates an example of the inheritance anomaly when using the behavior abstraction proposal [49]. The addition of a `get2` method, which atomically removes two items from the buffer, requires redefinition of both `get` and `put`. Redefinition of `get` and `put` is required because their behavior has changed to support `get2`. The functionality of `get` and `put`, however, has not changed. For example, in Figure 2.6, `get`'s "functionality" is `out++` and its "behavior" is the remaining lines.

Analysis of the inheritance anomaly in [55] reveals three causes:

- Partitioning of Acceptable States:
  Each object defines a set of acceptable states (e.g., empty, partial, and full in Figure 2.6) and uses synchronization constraints to ensure that valid transitions from one state to another. A specialization of a class through inheritance may introduce a method that further partitions the acceptable states of the object. Such a partitioning may introduce an inheritance anomaly if the language cannot partition the set of states without redefining the synchronization constraints of the parent class.

```
Activity subclass: #EnabledSetsActivity
    instanceVariableNames: 'enabledSelectors'

   "Parameter method for initialization of
        the activity."
   initialize
       super initialize.
       "Initialize the initial set of enabled
            selectors."
       enabledSelectors := self perform:
                   self initialAbstractStates

   "Return first message from the mailbox
        belonging to the current abstract state."
   nextMessage
       ^self mailBox
          firstMessageWithCondition:
             [:message |enabledSelectors includes:
                message selector]

   "Generic event method associated to
        completion of the invocation
        (computation)"
   eventComplete : aMessage
       super eventComplete: aMessage.
       "Abstract state transition: computing
          next set of enabled selectors"
       enabledSelectors :=
          self perform:
               (self nextAbstractStateAfter:
                   aMessage selector)
```

```
EnabledSetsActivity subclass:
    #BufferEnabledSetsActivity
    instanceVariableNames: ''

    "Abstract states."
    empty
       ^#(put:)

    full
       ^#(get)

    partial
    "Defined as the union (+) of empty
         and full."
       ^(self empty) + (self full)

    "The initial abstract state."
    initialAbstractState
       ^#empty

    "Abstract state transition"
    nextAbstractStateAfter: selector
       ^bself isEmpty
          ifTrue: [#empty]
          ifFalse: [bself isFull
                       ifTrue: [#full]
                       ifFalse: [#partial]]
```

Figure 2.5: Using Activity to synchronize a bounded buffer in Actalk.

- History-only Sensitiveness[1] of Acceptable States:

  A subclass that introduces a method for which valid invocations are history sensitive may introduce an inheritance anomaly if the parent class does not account for the required history in its instance variables. For example, if the parent class supports `get` and `put` methods and the subclass introduces a `gget` method that cannot be invoked immediately after a `put` invocation, then the subclass may be required to redefine the `get` and `put` methods to record invocation history.

[1]The term "sensitiveness" is taken from [55] and is used throughout this discussion, though "sensitivity" seems more appropriate.

```
Class b-buf : ACTOR
{
      int in, out, buf[SIZE];
   behavior:
      empty = {put};
      partial = {put, get};
      full = {get};
   public:
      void b-buf()
      {
         in = out = 0;
         become empty;
      }
      void put()
      {
         in++; // store an item
         if (in == (out + size)) become full;
         else become partial;
      }
      void get()
      {
         out++; // remove an item
         if (in == out) become empty;
         else become partial;
      }
}
```

```
Class x-buf : b-buf
{
      int in, out, buf[SIZE];
   behavior:
      x_empty = renames empty;
      x_one = {put, get};
      x_partial =
             {put, get, get2} redefines partial;
      x_full = {get, get2} redefines full;
   public:
      void x-buf()
      {
         in = out = 0;
         become x-empty;
      }
      void get2()
      {
         out += 2; // remove two items
         if (in == out) become x_empty;
         else if (in == out + 1) become x_one;
         else become x_partial;
      }
      void put()
      {
         in++; // store an item
         if (in == (out + size)) become x_full;
         else if (in == (out + 1)) become x_one;
         else become x_partial;
      }
      void get()
      {
         out++; // remove an item
         if (in == out) become x_empty;
         else if (in == out + 1) become x_one;
         else become x_partial;
      }
}
```

Figure 2.6: Inheritance Anomaly in Behavior Abstractions.

- Modification of Acceptable States:
  Invocation of a method defined in the subclass modifies the set of states under which the methods inherited from the parent class can be invoked. For example, imagine that the subclass adds a `lock` method and an `unlock` method. Once the `lock` method is invoked, no other method can be invoked until the `unlock` has been invoked.

Reference [55] analyzes proposals for resolving the inheritance anomaly, but each is shown to fail under at least one of the examples discussed above. Though the proposal by the authors of [55] successfully resolves the anomalous behavior demonstrated by the other proposals, it cannot resolve all history sensitiveness. The BALLOON language [11] and the CYES-C++ [66] language, however, successfully resolve the inheritance anomaly for all of the examples demonstrated in [55] and can seemingly resolve arbitrary history sensitiveness.

The complaints regarding the inheritance anomaly are focused on functionally unnecessary redefinition of methods and a violation of encapsulation. The proposals that seek to resolve the inheritance anomaly attempt to reduce the amount of redefinition required. Rather than requiring the redefinition of the implementation of a method, many of the proposals allow the redefinition, in some form, of the synchronization constraint. In fact, BALLOON [11] essentially defines a separate synchronization object that controls method invocation. However, the encapsulation argument does not seem to be resolved since the synchronization code in the subclass often takes advantage of knowledge of the synchronization code used in the parent class. Figure 2.7, adapted from [11], demonstrates the use of BALLOON to resolve an inheritance anomaly without modifying the implementation of the superclass. An integer queue type is defined and implemented along with its concurrency control object (CCIntQueue). The queue is extended to provide a `GetRight2` method that atomically removes two elements from the queue. This extension does not require any modification of the superclass to support the synchronization of the additional method. However, the `MayGet` method defined in the concurrency control portion of `IntQueue` returns an integer count of the number of items that may be removed from the queue rather than a boolean indicating whether or not items may be removed. The extension that provides the `GetRight2` method takes advantage of this convenient design decision to simplify the synchronization code required by the subclass.

CYES-C++ [66] also separates class definitions from synchronization implementations and uses a novel approach to defining event sets, a representation of interacting methods, and the interaction between event sets. Event sets are constructed from primitive event sets using method parameters, conditional event sets, named event sets, and set operations. An interaction specification is defined using event sets and event ordering constraint expressions. Figure 2.8, adapted from [67], demonstrates the specification of a synchronized queue and the mix-in of a lock object in CYES-C++. In this example, `QEmpty` and `QFull` represent the event sets that correspond to an empty and full queue, respectively. The `SeqPuts` and `SeqGets` interactions serialize puts and gets. `DelayPutIfFull` and `DelayGetIfEmpty` ensure that a put will be delayed if the queue is full and a get will be delayed if the queue is empty. The `LockObj` class defines interactions that guarantee that invocations belonging to the `LockEventSet` event set will be delayed if the object is locked. Finally, `LockedQueue` extends `LockEventSet` to delay `put` and `get` if the object is locked.

The inheritance anomaly does not seem to be fundamentally tied to concurrency but rather to any attempt to modify the default stack-based invocation semantics defined by the programming language. Any feature of a language (concurrency, security) that allows a parent class to modify the invocation semantics for a set of methods and then allows a subclass to further modify those semantics, without strict rules, will suffer an inheritance anomaly. The authors of [59] argue that an inheritance anomaly can arise due to exception handling. Specifically, an inheritance anomaly due to exception handling can occur if a subclass wants to specialize or handle an exception that a parent class method is signaling, or a virtual method is allowed to be redefined to signal a new exception that is not expected by any of the ancestor class methods calling the virtual method.

```
// type definition
Type IntQueue
{
   Integer PutLeft(Integer);
   Integer GetRight();
   Integer Size();

   CC:
      Integer MayPut(); Integer MayGet();
      Bool AllowLeft(); Bool AllowRight();
      Delay PutLeft Until
         (AllowLeft() && MayPut() > 0);
      Delay GetRight Until
         (AllowRight() && MayGet() > 0);
}

// implementation
Class CIntQueue Implements IntQueue
{
      // implementation code
}

// concurrency control
Class CCIntQueue Controls CIntQueue
{
   Data Integer(CInteger) mayput, mayget;
   Data Bool(CBool) allowleft, allowright;

   CCIntQueue()
   {
      allowleft := true;
      allowright := true;
      mayput := Size();
      mayget := 0;
   }
   Integer MayPut() { return mayput;}
   Integer MayGet() { return mayget;}
   Bool AllowLeft() { return allowleft;}
   Bool AllowRight() { return allowright;}
   Wrapper PutLeft { allowleft := false;
         mayput := mayput - 1;}
      Inner { allowleft := true;
         mayget := mayget + 1;}
   Wrapper GetRight { allowright := false;
         mayget := mayget - 1;}
      Inner { allowright := true;
         mayput := mayput + 1;}
}
```

```
// type definition
Type IntQueue2 SubtypeOf IntQueue
{
   IntegerPair GetRight2();

   CC:
      Delay GetRight2 Until
         (AllowRight() && MayGet() > 1);
}

// implementation
Class CIntQueue2 Implements IntQueue2
{
   // implementation code
}

// concurrency control
Class CCIntQueue2 Controls CIntQueue2
{
   Inherits CCIntQueue;

   Wrapper GetRight2
      { allowright := false;
         mayget := mayget - 2;}
      Inner
      { allowright := true;
         mayput := mayput + 2;}
}
```

Figure 2.7: Inheritance anomaly resolution in BALLOON.

```
concurrent class queue
{
   public:
     queue();
     ~queue();
     void put(char);
     char get();
     Boolean Full();
     Boolean Empty();
   interaction:
     AddQ = put;
     RemQ = put;
     QEmpty = get:Empty();
     QFull = put:Full();
     SeqPuts = Serialize(AddQ);
     SeqGets = Serialize(RemQ);
     DelayPutIfFull =
         Priority(AddQ, QEmpty);
     DelayGetIfEmpty =
         Priority(RemQ, QFull);
   private:
     ...
};
```

```
concurrent class LockObj
{
   public:
     lock();
     unlock();
     Boolean Locked();
   interaction:
     LockEventSet = lock:Locked();
     UnlockEventSet = unlock;
     LockObjExp(LockEventSet,
         UnlockEventSet);
};

concurrent class LockedQueue:
   public LockObj, queue
{
   interaction:
     LockEventSet =
          LockObj::LockEventSet +
          put:Locked() +
          get:Locked();
};
```

Figure 2.8: Inheritance anomaly resolution in CYES-C++.

# Chapter 3

# Java's Concurrency Support

Java [38] has proven to be a relatively clean and simple object-oriented programming language. Java takes a somewhat pure approach to object-oriented programming. All user-defined structures are objects, and every "function" must be written as a method field of a class (as either a class method or as an instance method). Java modules are defined as either interfaces or classes (concrete or abstract). An `interface` specifies a set of public methods that an implementing class must define. A `class` specifies the internal representation and behavior of its instances. A subclass extends another class or implements any number of interfaces. All instance method invocations in Java are dynamically dispatched (see Section 2.1). In addition to the core language, Java provides a rich set of libraries for general programming, concurrent programming, and introspection.

This chapter discusses the concurrent programming features of the Java programming language. In particular, this chapter presents Java's support for multithreaded programming, thread synchronization, remote method invocation, and message services. Support for these features (excluding some aspects of synchronization) is provided through class libraries and, in some cases, specialized code translators (e.g., `rmic` for remote method invocation (RMI)). Though the discussions of individual features highlight specific shortcomings, the chapter concludes with a general discussion of the shortcomings of Java's concurrency support.

## 3.1 Multithreaded Programming

In Java, threads are associated with objects and each thread executes a specific body of code. There are two techniques for defining the body of code that a thread will execute. Under the first technique, a class extends `java.lang.Thread` and defines a `run` method taking no arguments. When the `start` method is invoked on an instance of a subclass of `java.lang.Thread`, a thread is created to execute the `run` method defined by the subclass.

Under the second technique, a user-defined class, `C`, implements the `java.lang.Runnable` interface and defines a `run` method taking no arguments. An instance of the defined subclass, `C`, is passed to the constructor of `java.lang.Thread`. When the `start` method of the `java.lang.Thread` instance is invoked, a thread is created to execute the `run` method defined in the instance of `C`

### 3.1.1 Thread Scheduling

In the interest of portability and machine independence, the Java language specification allows Java threads to be implemented as either user-level or kernel-level threads. As such, the language specification does little to restrict or influence the scheduling or preemption of threads. Each thread has an associated priority. When competing for computing resources, "threads with higher priority are generally executed in preference to threads with lower priority" [38].

Thread scheduling can, potentially, be affected through the use of the `yield` and `sleep` methods defined in `java.lang.Thread`. A `yield` is intended as a hint to the scheduler to pick another thread to execute, but there is no guarantee that the scheduler will do so. A `sleep` will delay thread execution for a specified amount of time.

The lack of guarantees in thread scheduling requires that programmers use synchronization in the assumption that threads are preemptively scheduled. For some Java implementations, this synchronization will be unnecessary and costly, but for portability synchronization is required.

## 3.2 Thread Synchronization

Java provides a primitive monitor-like mechanism to support thread synchronization. Each object has an associated lock that is used to ensure mutual exclusion within an object. Java, however, does not permit direct access to an object's lock. Instead, an object's lock is acquired as part of the execution of a `synchronized` statement. Figure 3.1 demonstrates the use of a `synchronized` statement within the body of a method. When a thread executes the statement, it must acquire the lock of the specified object (`objOne`, in the example). Once the lock is acquired, it is held until the thread completes execution of the body of the `synchronized` statement.

```
...
Object objOne = new Object();
...
synchronized (objOne)
{
   // critical section
}
...
```

Figure 3.1: Thread synchronization.

A method within an object can be marked `synchronized` to ensure mutual exclusion during the entire execution of the method. When a thread invokes a `synchronized` method of an object, the thread must acquire the lock associated with the object[1]. After acquiring the lock, no other thread can execute a `synchronized` method within the same object or a `synchronized` statement that specifies the locked object.

Lock possession can be temporarily relinquished within a `synchronized` method or statement by invoking one of the `wait` methods on the object that has been locked. Invocation of a `wait` method causes the thread to wait until another thread invokes a `notify` method (or until an optional time limit expires). Each object supports two `notify` methods: `notify` and `notifyAll`. The basic `notify` method awakens a single arbitrary thread from the set of waiting threads, whereas the `notifyAll` method awakens all waiting threads. Once awakened (and after the notifying threads relinquishes the lock), the notified threads must compete with all other threads attempting to acquire the object's lock; the awakened threads enjoy no privileges in trying to acquire the lock.

[1]A `synchronized` static method must acquire the lock associated with the class.

The support for synchronized methods, waiting, and notifying amounts to a monitor-like synchronization mechanism, though some [17] contend that Java does not really support monitors because of its lack of requirements on the synchronization of methods. In particular, not all methods of an object need to be synchronized. As such, some methods may modify variables that are used within a critical section of a synchronized method. In addition, Java does not directly support multiple condition queues, though they can be emulated using additional lock objects.

An additional peculiarity of Java's thread synchronization support is the division of the support between the class library and the language. In particular, the locks associated with each object and the `wait` and `notify` methods are all inherited from the `Object` superclass. But, these locks cannot be directly manipulated through an object's interface. Locks are acquired and released only through `synchronized` methods and statements. Restricting lock manipulation to the use of `synchronized` enforces some discipline on the use of the locks and prevents mismatched locks and unlocks. Furthermore, the notion of threading exists only at the language-level in that the support for `synchronized` methods and statements implies the existence of multiple threads[2].

## 3.3 Remote Method Invocation

Java provides support for distributed programming using remote method invocation (RMI) [79]. Under RMI, "server" objects are created and remote references to these objects are provided to client programs (through a naming service, for example). Client programs then invoke methods supported by the remote "server" objects just as they would invoke methods on local objects. All argument marshaling and unmarshaling, and all network communication is hidden from the programmer.

To define a class for remote objects, the programmer defines an interface that extends `java.rmi.Remote` and that includes the methods that can be remotely invoked. Figure 3.2 defines the `Hello` interface with a single method, `sayHello`, that can be remotely invoked.

Next, a class is defined that implements the remote interface and that extends one of the predefined `RemoteObject` classes. Figure 3.3 defines a class that implements the `Hello`

[2]The language and virtual machine specifications do include discussions of threading relating to loose scheduling and memory accesses.

```
public interface Hello extends java.rmi.Remote
{
  String sayHello() throws java.rmi.RemoteException;
}
```

Figure 3.2: Remote interface.

interface and that extends `java.rmi.server.UnicastRemoteObject` to inherit support for receiving remote requests. This class defines a server program that, within the body of the `main` method, creates an instance of the implementing class, `HelloImpl`, registers the instance with the naming service (which must be running) using `Naming.rebind`, and waits for incoming invocations of method `sayHello`.

Client programs, such as that demonstrated in Figure 3.4, retrieve a remote reference to the RMI "server" objects and invoke methods through the server's interface. In the example given, the remote reference is retrieved from the same naming service with which the server registered. This reference is stored in a reference to the implemented interface and used as any other object reference.

When using RMI, the programmer must also use a special compiler (`rmic` in Sun's distribution) to generate the stubs that handle argument marshaling and network communication. In addition, the generated stubs must be made accessible to client programs so that they can be dynamically loaded. Java's RMI provides built-in support for loading stubs from URLs and allows programmers to implement custom class loaders that can be written to automatically retrieve stubs.

## 3.4 Message Service

Java 2 Enterprise Edition provides a message service API (JMS) [80] that allows point-to-point and publish/subscribe messaging. This support, however, is directed at enterprise solutions. As such, the message "destinations" exist outside of specific programs and are accessed through the JMS provided classes. Programs are provided, via some means, with the names of message destinations (in the point-to-point case, the name of a message queue). To use the messaging service, a program must create a connection to a specific destination before it can send or receive messages.

```
public class HelloImpl
  extends java.rmi.server.UnicastRemoteObject
  implements Hello
{
  public HelloImpl() throws java.rmi.RemoteException
  {
    super();
  }
  public String sayHello() throws java.rmi.RemoteException
  {
    return “Hello World!”;
  }

  public static void main(String args[])
  {
    try
    {
      HelloImpl obj = new HelloImpl();

      // Bind this object instance to the name “HelloServer”
      Naming.rebind(”//myhost/HelloServer”, obj);

      System.out.println(“HelloServer bound in registry”);
    }
    catch (Exception e)
    {
      System.out.println(“HelloImpl err: ” + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Figure 3.3: Remote implementation.

This messaging service is intended for loosely-coupled, enterprise-level solutions and is quite unwieldy for tightly-coupled concurrent programs. Message destinations are created and maintained using administrative tools and, as such, cannot truly be considered a language-level mechanism for distributed programming.

## 3.5 Shortcomings in Java's Concurrency Support

Java's concurrency model has two significant shortcomings:

- It provides remote objects with limited support for dynamic creation.

```
public class HelloClient
{
  // "obj" is the identifier used to refer to the remote
  // object that implements the "Hello" interface
  static Hello obj = null;

  public static void main(String [] args)
  {
    try
    {
      obj = (Hello)Naming.lookup("//" + getCodeBase().getHost() + "/HelloServer");
      System.out.println(obj.sayHello());
    }
    catch (Exception e)
    {
      System.out.println("HelloClient exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Figure 3.4: RMI client.

- It provides only a single synchronization primitive—remote method invocation—for distributed programming.

Standard Java/RMI's static virtual machine (VM) model allows a program to communicate with any number of remote objects. Each of these remote objects must be initialized prior to communication. These objects, as discussed in Section 3.3, are typically initialized through some external means (i.e., manual execution of a server program) often requiring a setup procedure before the program proper begins. This static model means that programs cannot adapt themselves to their input. For example, a distributed Java RMI program for computing Fourier coefficients (based on a sequential program in the Java Grande *Fourier* Benchmark[28]) might want to run on a varying number of processors. Unfortunately, that cannot be specified *easily or automatically* within a Java RMI program. (Although it is possible to write such a program in Java RMI, it requires manual interaction.)

Standard Java/RMI's only synchronization primitive for distributed programming is the remote method invocation. The limitations of languages that provide only one form of synchronization mechanism have been widely discussed in the literature [72, 2, 52, 34, 36].

As one example, having both synchronous and asynchronous message passing is desirable. Synchronous message passing (such as RMI) is very useful, especially for programming client/server interactions in a familiar style (e.g., procedure call syntax and semantics). However, asynchronous message passing is also useful. First, it can be used to avoid *remote delay* in which a server, in processing a request, invokes an operation in another server that might delay [52]. Asynchronous message passing can be used to invoke the remote operation whenever it is necessary for the first server to honor other requests in order to remove the conditions that led to remote delay. In a language that provides only synchronous message passing, extra processes must be employed to avoid remote delay; this often complicates problem solutions. Asynchronous message passing is also useful whenever it is not necessary to delay the invoker of an operation. For example, it can be used to program pipelines of filter processes, where it is most natural for the producer to continue after sending a message to the consumer.

Several languages that incorporate multiple synchronization primitives have been designed and implemented, e.g., Concurrent C [35], Lynx [73, 74], StarMod [25], and SR [4, 3]. Some work [8] has shown that an implementation of such a language (in [8], SR) can provide several synchronization mechanisms at a reasonable cost. Having multiple synchronization primitives proved useful in programming different upcall program structures [7]. Additional work [63] has shown that having asynchronous message passing is desirable — for simpler *and* faster code — even if a language provides a rendezvous mechanism.

# Chapter 4

# The SR Concurrency Model

The SR concurrent programming language [4, 3] provides a variety of mechanisms for writing parallel and distributed programs. The notions of virtual machines, resources, and operations are central to SR's concurrency model. *Virtual machines* (VMs) are used for distributing a program onto physical machines. Each VM resides on single physical machine, though many VMs may reside on the same physical machine. SR's primary modular component is the resource. Processes execute within a particular resource instance and have (shared) access to variables and operations within that instance. An operation can be considered a generalization of a procedure and enables processes to interact. VMs, resource instances, and operation instances are created dynamically. As such, SR uses *capabilities* to refer to each. A capability acts as a pointer and can be assigned to variables and passed as parameters, thus permitting, for example, dynamic communication paths.

This chapter gives an overview of the SR concurrency model, focusing, in particular, on operations, resources, and virtual machines. The chapter concludes with a discussion of additional concurrency features supported by SR. The presentation in this chapter is based on that in [3].

## 4.1 Operations

The notion of operations is the key abstraction that defines SR's concurrency model. An operation is a generalization of a procedure. Like a procedure, an operation has a name, can take parameters, and can 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

```
in getforks(id) st not eating[(id mod n) + 1] and not eating[((id-2) mod n) + 1] ->
     eating[id] := true
[] relforks(id) ->
     eating[id] := false
ni
```

Figure 4.1: Example `in` statement.

statement. An operation can be serviced in two ways: by a procedure or by `in` statements. (SR's `in` statement combines and generalizes aspects of Ada's accept and select statements.) This yields the following four combinations:

| *Invocation* | *Service* | *Effect* |
|---|---|---|
| **call** | **proc** | procedure call |
| **call** | **in** | rendezvous |
| **send** | **proc** | dynamic process creation |
| **send** | **in** | message passing |

A guard on an `in` statement can also contain a *synchronization* expression and a *scheduling* expression. The former specifies which invocations are acceptable; the latter specifies the order in which to service acceptable invocations. To illustrate, consider the `in` statement in Figure 4.1 which is taken from a dining philosophers solution. The first arm of the `in` statement uses a synchronization expression, introduced by `st` (such-that), to accept an invocation of `getforks` from a philosopher only when that philosopher's neighbors are not eating. Note how the synchronization expression references the invocation's parameter, `id`.

Operations provide tremendous flexibility in defining dynamic communication paths between cooperating processes. Operations may be passed to and returned from procedures using operation capabilities. A procedure may be invoked one time with an operation serviced by a procedure as an argument, and the next time with an operation serviced by an input statement.

## 4.2 Resources

The SR programming language is not object-oriented (see Section 2.1), but it is object-based (i.e., it lacks inheritance and dynamic dispatch). SR supports encapsulation of data and behavior in dynamic, object-like modules called *resources*. Resource instances are created via the `create` statement, which returns a resource capability for the newly created instance. A resource capability contains operation capabilities for each operation defined in the resource's specification; their values are assigned to those operations in the newly created resource instance.

```
resource Main()
   import Philosopher, Servant
   var n, t: int
   read(n); read(t)
   # create the Servant and Philosophers
   var s: cap Servant
   s := create Servant(n)
   fa i := 1 to n ->
     # (returned resource cap not needed)
     create Philosopher(s, i, t)
   af
end
```

Figure 4.2: Main resource in multiple-resource Dining Philosophers.

As an example, Figures 4.2 and 4.3 present a version of Dining Philosophers. It employs centralized control, but it places philosophers and servants in different resource instances. The main resource consists entirely of "initial" (top-level) code. This code is executed when an instance of the resource is created; in the case of `Main`, its initial code is executed when the program begins execution. `Main`'s initial code creates one instance of the `Servant` resource and `n` instances of the `Philosopher` resource, passing to each philosopher its integer identity (`i`), the capability for the servant (`s`), and the number of iterations it should execute (`t`).

As demonstrated in Figure 4.3, each philosopher requests forks from the servant, eats, and later releases forks. Forks are requested and released through synchronous invocations of the respective operations defined by the `Servant` resource. The `Servant` services invocations of its operations using an input statement. The input statement services a `getforks` request from a philosopher only when neither of the philosopher's neighbors are holding the forks. The input statement services `relforks` requests without restrictions.

```
resource Philosopher
    import Servant
body Philosopher(s: cap Servant; id, t: int)
    process phil
      fa i := 1 to t ->
         # get forks from Servant instance s
         s.getforks(id)
         # eat
         write("Philosopher", id, "is eating")
         # release forks back to Servant
         # instance s
         s.relforks(id)
         # think
         write("Philosopher", id, "is thinking")
      af
    end
end
```

```
resource Servant
    op getforks(id: int) # these ops are invoked
    op relforks(id: int) # by Philosophers
body Servant(n: int)
    process server
      var eating[1:n] := ([n] false)
      do true ->
         in getforks(id)
          st not eating[(id mod n) + 1]
          and not eating[((id-2) mod n) + 1] ->
              eating[id] := true
         [] relforks(id) ->
              eating[id] := false
         ni
      od
    end
end
```

Figure 4.3: Philosopher and servant resources in multiple-resource Dining Philosophers.

## 4.3 Virtual Machines

As noted previously, SR defines the notion of a *virtual machine* (VM) as its language mechanism for distributing a program onto physical machines. Like resource instances, VMs are created dynamically via the `create` statement. The program in Figures 4.2 and 4.3 executes on a single virtual machine and, therefore, on a single physical machine. It can be easily modified, though, so that each philosopher executes on a different virtual machine, each on a different physical machine. Only `Main`'s loop needs to be changed, for example, as shown in Figure 4.4. The `on` clause in the first `create` specifies the physical

```
fa i := 1 to n ->
    var vmcap: cap vm
    vmcap := create vm() on host[i]
    create Philosopher(s, i, t) on vmcap
af
```

Figure 4.4: Modified `Main` loop for multiple-VM Dining Philosophers.

machine on which the VM is to be created; the value returned by this `create` statement is a capability for the newly created virtual machine. The `on` clause in the second `create` specifies the virtual machine on which the resource instance is to be created. The array `host` contains physical machine names (as strings; its initialization is not shown).

## 4.4 Additional Features

SR provides abbreviations for commonly occurring uses of operations. For example, the `process` keyword (used in Figure 4.3) is defined as an abbreviation for an operation declaration, a proc, and one or more sends to that proc by the resource's initial code. Additionally, a `receive` statement is an abbreviation of a simple form of `in` statement, and semaphore declarations and `P` and `V` are abbreviations for simple operation declarations and `send` and `receive` statements.

SR also provides a few other statements that deal with invoking or servicing operations. SR's concurrent invocation statement provides a way to start several invocations at the same time; it terminates when all its invocations have completed. SR's `reply` statement allows a servicing process to send an "early reply" (and any return value) to its invoker, after which both the invoking and servicing processes continue executing. Figure 4.5 demonstrates the use of a `reply` statement. Notice that procedure `foo` continues to execute concurrently with its invoker after replying.

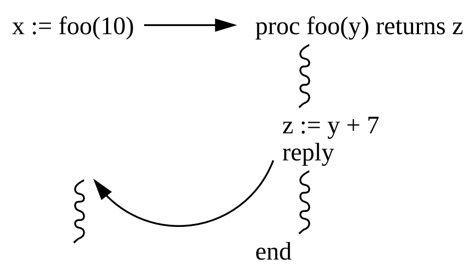



Figure 4.5: Early reply.

SR's `forward` statement defers replying to an invocation and instead passes this responsibility on to another operation. Figure 4.6 demonstrates the control flow of a program that executes a `forward` statement. Notice that procedure `foo` executes concurrently with procedure `goo` after the `forward` statement, and that procedure `goo` returns control (and any return value, `goo`'s `z` in the figure) to the original invoker of procedure `foo`, thus allowing the invoker to continue executing.

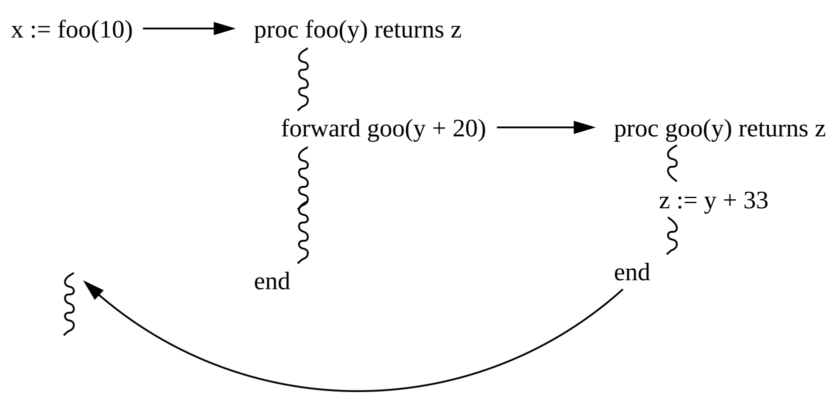



Figure 4.6: Forward.

# Chapter 5

# The JR Programming Language

JR is a concurrent object-oriented programming language that provides support for writing tightly-coupled concurrent programs and libraries. The driving forces behind the JR programming language are the recognition of the flexibility provided by the SR concurrency model (see Chapter 4) and the desire to adapt and extend SR's concurrency model (and the abstractions it provides) by integrating it with an object-oriented programming language (see Section 2.1). In particular, the generalization of procedures into operations allows a program to dynamically change the flow of communication between cooperating components. Moreover, since the manner in which an operation is serviced is not visible to clients of the operation, changing the servicing of an operation is transparent to the client.

For the design and implementation of JR, the Java programming language was selected as the "host" object-oriented language. As such, JR is the result of integrating the SR concurrency model with the Java programming language. Java is a relatively clean and simple object-oriented language that supports exception handling, garbage collection, platform-independent programming, and that provides a large class library. Nonetheless, as discussed in Section 3.5, Java provides limited support for concurrent programming.

Our overall approach extends the Java syntax, rather than augmenting Java with library calls for synchronization. This approach has potential advantages because synchronization is represented as actual language primitives [15, 74]. This higher-level approach can reduce development time for applications, increase portability, lead to better optimizations, and simplify reasoning about programs (verification). For example, by extending Java's syntax to support operations directly, the compiler can statically type-check the uses of the operations.

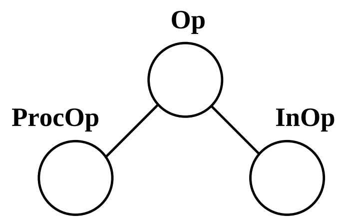



Figure 5.1: General JR Operation Inheritance Hierarchy.

JR provides SR-like operations in a novel object-oriented fashion. Figure 5.1 shows the general inheritance hierarchy of operation classes. As indicated in the figure, an operation is represented by the base class Op; this base class declares abstract methods for invoking and servicing an operation (`call`, `send`, and `inni`[1]). The two possible ways to service invocations — via a method or via `inni` statements — are, then, two classes, ProcOp and InOp, derived from the Op base class. The ProcOp and InOp classes define the methods for invoking and servicing an operation. For example, the InOp class defines these methods such that their actions apply to a queue of invocations declared local to the class. Conceptually (as well as in the implementation, see Section 5.3), the JR statements for invoking and servicing an operation are translated to invocations of methods in the operation's class.

As a simple example of the abstractions that JR provides over Java, Figure 5.2 gives a JR implementation of the RMI program presented in Section 3.3. In this example, a new virtual machine is created on a physical machine specified by a command-line argument. A new remote server object is then created on the new virtual machine. The client (here, the main method) invokes the `sayHello` operation in the remote object and prints the result. As this example demonstrates, JR relieves the programmer from dealing directly with the RMI registry, from writing separate server programs for remote objects, and from explicitly executing such server programs on different machines. Instead, remote objects (and the virtual machines in which they exist) are dynamically created during the execution of a JR program.

This chapter describes the concurrent programming features provided by the JR programming language, presents an overview of the implementation of these features, and gives performance results for microbenchmarks. As a running example, a simple solution to the Readers/Writers problem is given.

[1]The `inni` statement is JR's counterpart to SR's `in` statement

```
public class Hello
{
  public op String sayHello();
  public String sayHello()
  {
    return "Hello World!";
  }

  public static void main(String [] args)
  {
    try
    {
      // create the Hello server on a new virtual machine
      remote Hello obj = new remote Hello() on (new vm() on args[0]);
      System.out.println(obj.sayHello());
      JR.exit(0);
    }
    catch (Exception e)
    {
      System.out.println("Hello exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Figure 5.2: JR Equivalent to Java RMI Example.

## 5.1 Concurrent Programming Features in JR

JR adapts the following features from SR (see Chapter 4): dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, and asynchronous message passing. These features have been integrated with Java in an attempt to retain the "feel" of Java while providing a richer concurrency model. In JR, Java classes take the place of SR resources and Java methods take the place of SR procs.

### 5.1.1 Dynamic Remote Virtual Machine Creation

A JR program begins by executing the `main` method within the specified (executed) "main" class. The `main` method is executed within an initial virtual machine created on the physical host on which the program is executed. Additional virtual machines can be dynamically created to support the execution of multiple virtual machines on a physical host and to support the distribution of virtual machines among many physical hosts. Potentially remote virtual machines are created using Java's "new" expression. Virtual machine creation takes the following general syntactic form:

`vm` <vmName> = `new vm()` [ `on` <hostName> | `on` <vmVar>];

The optional "on" clause is used to specify the host on which the new virtual machine is to be created. This host can be specified as either a Java `String`, in which case the new virtual machine is created on the host specified by the string, or as a reference to another virtual machine, in which case the new virtual machine is colocated with the specified virtual machine. If an "on" clause is not specified, then the virtual machine is created on the physical machine of the instantiating process. Line 12 in Figure 5.3 demonstrates the creation of a virtual machine to house the server object in the Readers/Writers solution.

### 5.1.2 Dynamic Remote Object Creation

JR provides support for populating remote virtual machines with new objects through dynamic remote object creation. Remote objects are created using Java's familiar "new" expression. A reference to a remote object in JR is stored in a `remote` reference rather than a standard Java reference. Remote object creation takes the following general syntactic form:

`remote` <class name> <var name> = `new remote` <class name> (<actuals>) [`on` <vmVar>];

An example of the creation of the remote Readers/Writers server can be seen on line 13 in Figure 5.3.

A remote object reference provides the interface through which a remote object is manipulated. This interface is the set of operations defined by the object's class, e.g., lines 3-7 in Figure 5.5. Operations are discussed in Section 5.1.3. A remote interface does not include the methods defined within a class. As such, remote instances of classes that *do not* define operations provide no interface for interaction (i.e., such instances cannot receive invocations). Such classes include all of the classes provided in the Java class library. This limitation arises because JR draws a distinction between methods and operations and because the Java class library has not been modified to support operations.

```
public class rwMain
{
  public static void main(String [] args)
  {
    op void done(); // create a local op

    // parse command line arguments to determine
    // #readers, #writers and Server destination host
    ...

    // create on specified host
    vm servVM = new vm() on args[0];
    remote Server serv = new remote Server() on servVM; // remote object creation

    for (int i = 0; i < readers; i++)
        new Reader(r_iters, serv, done);
    for (int i = 0; i < writers; i++)
        new Writer(w_iters, serv, done);

    int waitfor = readers + writers;

    // wait for each R and W to signal completion
    while (waitfor > 0)
    {
        receive done(); // abbreviated “in” statement
        waitfor--;
    }
  }
}
```

Figure 5.3: Readers/Writers main class in JR.

### 5.1.3 Operations

The operation is the key abstraction that forms the basis for the SR (and JR) concurrency model. In JR, an operation is an abstraction of a method[2]. Operations are declared, like methods, as part of a class definition. JR supports both class and instance operations. As mentioned above, a remote object's interface is defined by the set of operations declared in the object's class. Operation declaration takes the following general syntactic form:

<modifiers> `op` <return type> <opname> (<formals>) [exceptions];

[2]JR supports both operations and methods, but treats them differently as discussed in Section 5.2.4.

```
1   public class Reader
2   {
3     int iters;
4     remote Server serv;
5     cap void () done;
6
7     protected op void start();
8
9     public Reader(int iters, remote Server serv, cap void () done)
10    {
11      this.iters = iters;
12      this.serv = serv;
13      this.done = done;
14      send start(); // start the client thread
15    }
16
17    protected void start()
18    {
19      while (iters > 0)
20      {
21          iters--;
22          call serv.readRequest();
23          ... // read
24          call serv.readRelease();
25      }
26      // tell main this thread has finished
27      send done();
28    }
29  }
```

Figure 5.4: Reader class (Writer is similar).

An operation definition consists of a declaration and an implementation. An operation's declaration defines the signature (i.e., formal parameter types, return type, and exception types) of the operation and adds the operation to the specification of the class.

An operation's implementation is defined by either a method with a matching signature or a set of input (`inni`) statements that service the operation. An operation implemented by a method is called a ProcOp. Each invocation of a ProcOp is "serviced" by executing the body of the method implementing the ProcOp. In Figure 5.5, for example, the `start` operation declared on line 7 is implemented by the method defined beginning on line 14.

An operation implemented by a set of `inni` statements is called an InOp. By

```
1   public class Server
2   {
3     public op void readRequest();
4     public op void readRelease();
5     public op void writeRequest();
6     public op void writeRelease();
7     protected op void start();
8
9     public Server()
10    {
11      // create a thread to execute the start method
12      send start();
13    }
14    protected void start()
15    {
16      int nw = 0, nr = 0;
17
18      // on each iteration, service
19      // a R or W request or release
20      while (true)
21      {
22          inni void readRequest() st nw==0  nr++;
23          [] void writeRequest() st nw==0 && nr==0  nw++;
24          [] void readRelease()  nr--;
25          [] void writeRelease()  nw--;
26      }
27    }
28  }
```

Figure 5.5: Readers/Writers Server in JR.

default, an operation that is declared without a corresponding method is considered to be an InOp. The declaration of an InOp implicitly defines an implementation that consists of an invocation queue. When an InOp is invoked, an invocation is placed in the operation's queue until an `inni` statement services the invocation. Each invocation is only serviced by a single `inni` statement, which executes the body of code that corresponds to the arm servicing the operation. As such, the actual implementation of an InOp is provided by a *set* of `inni` statements that service the operation. In Figure 5.5, for instance, the `inni` statement on lines 22–25 services invocations of the operations defined in lines 3–6.

JR also allows operations to be declared abstract with the restriction that the defining class be declared abstract. An abstract operation is, by definition, neither an InOp

```
abstract class A
{
  public abstract op void foo();
  public abstract op void bar();
}
```

(a) Abstract class.

```
class B extends A
{
  public op void foo();

  public op void bar();
  public void bar()
  { inni void foo() { ... } }
}
```

(b) Concrete subclass.

Figure 5.6: Concrete redefinition of abstract operations.

nor a ProcOp since these labels are derived from the operation's implementation and an abstract operation has no implementation. The implementation of an abstract operation is eventually defined in a subclass (making the operation concrete) just as the implementation of an abstract method in Java is defined in a subclass. Figure 5.6 demonstrates the definition of two abstract operations, in Figure 5.6 (a), that are made concrete in the extending class, in Figure 5.6 (b). Operation `foo` is implemented as an InOp by the subclass whereas operation `bar` is implemented as a ProcOp. The redefinition of operation through inheritance is discussed in Section 5.2.

#### 5.1.3.1 Operation Equivalence Classes

An InOp can be serviced by a number of input statements simultaneously executed by a number of threads. These threads must compete for access to the operation queues of each InOp appearing in the input statements. To identify and control conflicts in accesses to operation queues by competing threads, operations are grouped into *equivalence classes* based on their actual use in input statements. These equivalence classes are dynamically formed during the execution of a program. Each operation initially forms its own equivalence class. When operations appear in an input statement together, their equivalence classes are merged. All operation locking is done at the level of the equivalence class. JR's equivalence class implementation places no restrictions on the operations and operation capabilities that may appear in the same input statement. SR, for example, requires that an operation capability referring to an operation in a different resource appear alone in an input statement (i.e., there can be no other arms). Further comparison of JR's equivalence classes and SR's equivalence classes can be found in [64].

### 5.1.4 Operation Capabilities

Operations can be passed as arguments to methods, returned as results from methods, and assigned to variables through the use of operation capabilities. This flexibility allows programs to define dynamic communication paths between interacting components. An operation capability is simply a reference to an operation and is declared as follows:

`cap` <return type> (<formal types>) [<exception types>] <variable list>;

Recall that lines 3–7 in Figure 5.5 demonstrate the declaration of the set of operations supported by the server object in the Readers/Writers example. An explicit, although simple, use of an operation capability can be seen on line 27 in Figure 5.4. This capability stores a reference to the done operation defined on line 5 of the main class in Figure 5.3. The done operation is used by each of the reader threads and each of the writer threads to notify the main class that the thread has completed.

#### 5.1.4.1 Variance

An operation capability may refer to any operation with a matching signature. Support for variance in operation references was explored and, finally, rejected due to typing issues. Early versions of the JR compiler allowed an operation capability to refer to an operation that is contravariant in the argument types and covariant in the return type (for a discussion of variance in subtyping, see [70]) and that throws exceptions that are a subtype subset of the operation capabilities exceptions.

Unfortunately, supporting such variance allows the type-checker to permit programs that fail with illegal casting exceptions during message reception. Consider the example in Figure 5.7, in which an operation, `theOp`, is defined to take, as an argument, an `Object` reference. The operation is assigned to an operation capability, `theCap`, under the support for variance just discussed. This assignment is legal because any invocation of the capability will require a `String` object as an argument and every `String` object is also a `Object` object. Next, the operation is asynchronously invoked and passed an `Integer` object. This invocation is perfectly valid because the operation expects objects of type `Object`, which all `Integer` objects are. The problem arises when the program attempts to service the invocation through the use of the operation capability. An invocation with an argument of type `String` is expected, but an invocation (the one just sent) with an argument of type `Integer` is received. An object of type `Integer` cannot be implicitly into an object of type `String`, so an illegal cast exception is raised.

```
op void theOp(Object);
cap void (String) theCap;

theCap = theOp; // valid assignment under cap/op variance

send theOp(new Integer(3)); // valid send

inni void theCap(String s) // erroneous receive
{
  System.out.println("Got it");
}
```

Figure 5.7: Erroneous program under support for operation capability variance.

The reason for this erroneous behaviour is the manner in which the variance is defined for operation capabilities. The variance discussed above is perfectly valid for invoking operations through capabilities, but the opposite variance for the argument types is required for servicing (receiving from) operations through capabilities (i.e., the operation should be covariant in the argument type) to guarantee type compatibility. For example, in Figure 5.7, sending is safe under the semantics that `theCap` and `theOp` are contravariant in their argument types, but `theCap` would need to receive into a supertype of `theOp`s argument type to safely receive anything passed to `theOp`. As such, a capability and the operation assigned to it must be both contravariant *and* covariant in their argument types (i.e., invariant).

A modification of the semantics for the input statement can enable safe support for the originally explored variance, but doing so would complicate both the semantics and the implementation. For example, the semantics could be extended to only allow reception of an invocation if the types were compatible. This extension allows the example code to execute safely, but limits the utility of operation capabilities because some invocations become implicitly unserviceable by the input statement. Thus far, the originally explored variance has only been found to be useful or desirable in artificial examples, none of which have provided compelling reasons for its support.

#### 5.1.4.2 Overloaded Operations

Unlike SR, JR allows overloaded operations in much the same way that Java allows overloaded methods. If two operations within a class (either declared or inherited) have the same name but differ in the number or types of their parameters, then the operations are overloaded. Typically, when an overloaded operation (or method) is invoked, the number of actual arguments and the types of these arguments are used to determine which instance of the overloaded operation (or method) is invoked. In JR, however, this resolution mechanism is complicated by the fact that operations and methods can accept operations, which themselves may be overloaded, as arguments. JR extends Java's resolution mechanism to handle the resolution of overloaded operations (or method) invoked with overloaded operations as arguments.

```
1   static void foo(cap void(int) c)
2   { System.out.println("foo int"); }
3
4   static void foo(cap void(long) c)
5   { System.out.println("foo long"); }
6
7   static op void f(int);
8   static op void f(long);
9
10  public static void main(String [] args)
11  {
12     foo(f); // prints foo long
13     foo((cap void(int)) f); // prints foo int
14  }
```

Figure 5.8: Invocation of an overloaded method with an overloaded operation as an argument.

Consider the invocation of the overloaded method `foo` with the overloaded operation `f` as its argument on line 12 of Figure 5.8. The resolution mechanism first determines the most specific definition of `foo` for which any definition of `f` is a valid argument (i.e., type compatible). For the invocation on line 12, the selected definition of `foo` is that starting on line 4 because `cap void (long)` is more specific than `cap void (int)`. Next, the most specific definition of `f` that can be passed as an argument to the resolved `foo` is selected. In the invocation on line 12, the `f` operation defined on line 8 is passed as the argument to `foo`. The selection strategy can be modified through the use of explicit casts and, of course, the use of operation capabilities (which are uniquely named). Line 13 in Figure 5.8 demonstrates the use of an explicit cast to invoke the `foo` method defined on line 1 with the `f` operation defined on line 7 as its argument. The explicit cast is used to "select" the desired `f` operation and to restrict the potential matches for `foo`.

### 5.1.5 Asynchronous Message Passing

Asynchronous communication is supported via the `send` statement. A `send` is very similar to a normal method invocation, but requires the `send` keyword and cannot be used as part of an expression (i.e., an asynchronous method invocation in JR is a statement). After executing a `send` statement, the invoking thread continues to execute without waiting for the invoked operation to complete. Again, methods and operations are distinct in JR and, as such, only operations can be asynchronously invoked.

If an operation invoked by `send` is serviced by a method (i.e., the operation is implemented by a method), then a new thread is created to execute the method. If the operation is serviced by an `inni` statement, then a message is created to store the arguments of the invocation. This message is then added to the invocation queue for the operation.

## 5.2 Operations and Inheritance

Inheritance in Java is discussed in Chapter 3, but is briefly discussed again here to highlight some important points for the ensuing discussion of operation inheritance. A Java class definition consists of a specification and an implementation (a Java interface consists of only a specification). The specification of a class defines the external interface "exported" by instances of the class. An instance of a class is used by invoking instance methods defined in the class' specification. The implementation of a class defines the actions taken when a method in the specification is invoked (i.e., the implementation defines the statements that are executed upon invocation).

In Java, a new class may be derived from an existing class to create a subclass. Through this derivation, the subclass inherits the specification of its parent class. By default, a subclass also inherits its parent's implementation of the specification. A subclass can extend the inherited specification through the addition of methods. Similarly, a subclass can modify the implementation of its inherited specification by redefining the inherited methods.

### 5.2.1 Operation Inheritance

In JR, a derived class may modify the implementation of its inherited specification by redefining the implementation of its inherited methods and operations. An inherited method's implementation is modified, as in standard Java, by redefining the method. In general, JR allows a subclass to redefine the implementation of an inherited operation as either a ProcOp or an InOp, regardless of the operation's implementation in the superclass. Redefinition of an operation's implementation requires an explicit redeclaration of the operation in the subclass only if the redefinition changes the operation from an InOp to a ProcOp or vice-versa. Otherwise, an explicit redeclaration of the operation is not required.

The notation $\mathrm{Op}_1 \rightarrow \mathrm{Op}_2$ means that the superclass defines the operation as an $\mathrm{Op}_1$ and the subclass is redefining the operation to be an $\mathrm{Op}_2$.

1. ProcOp $\rightarrow$ ProcOp

A redefinition from a ProcOp to a ProcOp corresponds directly to a method redefinition in standard Java. The subclass can simply redefine the method associated with the operation. Such a redefinition allows a subclass to specialize the operation implementation.

2. InOp $\rightarrow$ InOp

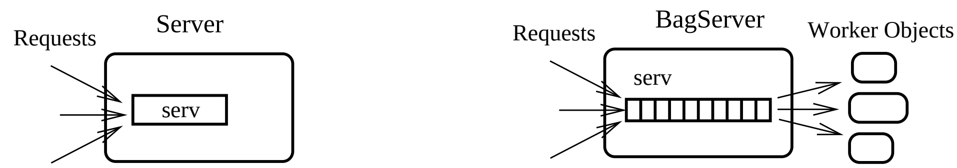
The implementation of an InOp is not actually redefined but rather extended. Any `inni` statements that "service" an inherited InOp are added to the set of `inni` statements that implement the operation. A subclass may explicitly redeclare an InOp as an InOp by explicitly redeclaring the operation. This redeclaration allows a subclass to relax access restrictions but does not create a separate invocation queue.

3. ProcOp $\rightarrow$ InOp

A ProcOp may be redefined as an InOp in a subclass by explicitly redeclaring the operation and not defining a signature-compatible method. The signature-compatible method that would have been inherited from the superclass is ignored.

4. InOp $\rightarrow$ ProcOp

An InOp may be redefined as a ProcOp in a subclass by both redeclaring the operation and defining a signature-compatible method.




(a) Centralized server. (b) Distributed server.

Figure 5.9: Distribution of servicing through redefinition of operation in subclass BagServer.

### 5.2.2 Distributing Operation Servicing through Redefinition

Redefinition of a ProcOp to be an InOp can be used to distribute the servicing of the operation's invocations without changing the client. Figure 5.9 graphically depicts both the original client-server structure, which uses a ProcOp, and the new bag of tasks structure that results from redefining the inherited operation to be an InOp. With the original operation, each invocation results in a new thread being created (at the server) to service the invocation.

With the redefined operation, each invocation is handled by an extant Worker object, which was created at program startup and which may be located on a separate host. Figure 5.10 shows the definition of the subclass `BagServer` and the redefinition of the operation. `BagServer` redefines the operation `serv` to be an InOp (by redeclaring the operation without defining a signature-compatible method), and creates Worker objects, passing a capability for `serv` to the constructor of each Worker object. Each Worker object repeatedly executes an `inni` statement to service invocations on `serv`'s queue. These Worker objects can be located on an arbitrary set of physical machines as specified by `hosts`.

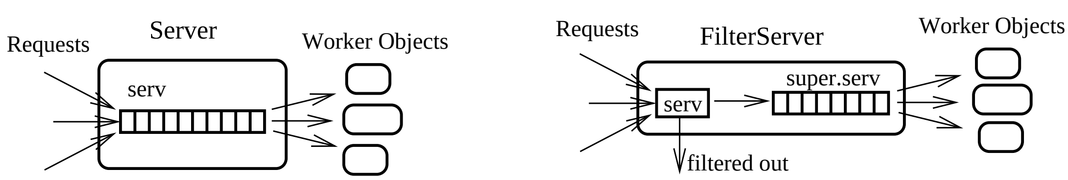
### 5.2.3 Filtering Operation Servicing through Redefinition

It is possible to filter the invocations of an operation by redefining an InOp to be a ProcOp. Figure 5.11 graphically depicts each server configuration. With the original operation, as shown in Figure 5.11 (a), each invocation is serviced by an extant Worker object. Redefinition of the operation, as shown in Figure 5.11 (b), allows for the filtering of invocations in order to reduce the amount of work done by the Worker objects. Figure 5.12 shows the definition of the subclass `FilterServer` that redefines an InOp to be a ProcOp. `Server` defines the `serv` operation as an InOp on line 4. `FilterServer` redefines the `serv` operation to be a ProcOp on lines 38–45.

```
public class Server
{
  // operation declaration
  public op int serv(int i);

  // signature-compatible method
  public int serv(int i)
  {
    // compute and return value
  }
}

public class Worker
{
  cap int (int) srvr;
  public Worker(cap int (int) srvr)
  {
    this.srvr = srvr;
  }
  process work()
  {
    while (true)
    {
      inni int srvr(int i)
      {/* compute and return value */}
    }
  }
}
```

```
public class BagServer extends Server
{
  // redeclaration of operation as InOp,
  // no signature-compatible method
  public op int serv(int i);

  public BagServer(vm [] hosts)
  {
    // initialize Worker objects
    for (int i = 0; i < hosts.length; i++)
    {
      remote Worker w =
        new remote Worker(serv)
          on hosts[i];
    }
  }
}
```

Figure 5.10: Redefining an operation to distribute its implementation.




(a) Distributed server.

(b) Filtered distributed server.

Figure 5.11: Filtering of invocations through redefinition of operation in subclass FilterServer.

Each invocation of `serv` defined in the subclass is routed through the method associated with the ProcOp to determine whether or not the invocation will be passed on to a Worker object. If the invocation is not rejected by the filter, then the subclass

```
1   public class Server
2   {
3     // operation declaration
4     public op long serv(int i);
5
6     public Server() {}// empty constructor
7     public Server(vm [] hosts)
8     {
9       // initialize Worker objects
10      for (int i = 0; i < hosts.length; i++)
11      {
12        remote Worker w =
13          new remote Worker(serv)
14            on hosts[i];
15      }
16    }
17  }
18
19  public class Worker
20  {
21    cap long (int) srvr;
22    public Worker(cap long (int) srvr)
23    {
24      this.srvr = srvr;
25    }
26    process work()
27    {
28      while (true)
29      {
30        inni long srvr(int i)
31        {/* compute and return value */}
32      }
33    }
34  }
35  public class FilterServer extends Server
36  {
37    // redeclaration of operation
38    public op long serv(int i);
39    public long serv(int i)
40    {
41      if (filter(i))
42        return DEFAULT; // or exception
43      else // forward to Worker objects
44        forward super.serv(i);
45    }
46
47    boolean filter(int i)
48    { // simple filter
49      return (i < 0);
50    }
51
52    public FilterServer(vm [] hosts)
53    {
54      super();
55      // initialize Worker objects
56      for (int i = 0; i < hosts.length; i++)
57      {
58        remote Worker w =
59          new remote Worker(super.serv)
60            on hosts[i];
61      }
62    }
63  }
```

Figure 5.12: Redefining an operation to serialize invocations through a filter.

uses a `forward` statement to pass responsibility for servicing the invocation to the InOp `serv` defined in the parent class (`Server`). Each Worker object repeatedly executes an `inni` statement to service the InOp `serv` defined in the `Server` class; this is the operation capability that is passed to the constructor of the Worker object on line 59 in Figure 5.12.

### 5.2.4 Redefinition Considerations

JR requires that a subclass explicitly redeclare an operation if the subclass redefines an inherited InOp as a ProcOp or an inherited ProcOp as an InOp. Such a redeclaration is required to statically determine that the operation has been redefined and to reduce the potential for erroneous code. Such a redeclaration is required as a statement of intent and an aid to the compiler for static analysis.

Imagine that an explicit redeclaration were not required to redefine a ProcOp as an InOp. Instead, assume that an operation is implicitly redefined as an InOp if the operation is serviced by an `inni` statement. Full program analysis would be required to statically determine that the operation has been redefined. However, such an analysis is not sufficient when capabilities are used within `inni` statements. This "redefinition" approach is unsatisfactory because all such redefinitions cannot be discovered until run-time. Furthermore, allowing such a "redefinition" could lead to hard-to-find errors if a programmer accidentally "redefines" the wrong operation in an `inni` statement.

An InOp may be redefined to be a ProcOp if the subclass explicitly redeclares the operation and defines a signature-compatible method. If an `inni` statement attempts to service the operation through a reference to the subclass, then a compile-time error will be raised (see Figure 5.13 (b), line 6). If an `inni` statement attempts to service the operation through a reference to the superclass, then a run-time error will be generated (see Figure 5.13 (b), line 8). This explicit redeclaration is required to reduce the potential for accidental redefinitions and to make the change in invocation servicing semantics explicit.

JR makes a strict distinction between operations and methods. An inherited operation may not be redefined as a method and an inherited method may not be redefined as an operation. This distinction is required because the concurrency constructs introduced in JR (e.g., the *send* statement) apply only to operations and capabilities and it is desirable to statically check improper use. If a subclass is allowed to redefine an inherited operation as a method, then a *send* to that method will fail at run-time because the method will not have been properly translated to support asynchronous invocation. Moreover, since the set of operations defined in a class directly translates to the interface exported by remote instances of that class (see Section 5.1.2), allowing a standard method to redefine an operation is equivalent to removing the operation from the remote specification. As such, this restriction is consistent with the behavior of Java because a subclass is not permitted

```
class A
{
   public op void foo();
}

class B extends A
{
   public op void foo();
   public void foo()
    { ... }
}
```

(a) Redefinition of InOp as ProcOp.

```
{
   ... // use of operations
   A a = new B();
   B b = new B();

   inni void b.foo() // compile-time error
                     // b.foo is a ProcOp
   { ... }
   [] void a.foo() // run-time error
                   // a refers to a B object
    { ... }
}
```

(b) Erroneous uses.

Figure 5.13: Erroneous uses of ProcOps in an `inni` statement.

to remove declarations from its inherited specification.

Redefining an inherited method to be an operation is not supported for implementation reasons. An operation is implicitly defined to include the RemoteException exception in its throws clause. This inclusion is to support the use of operations as interfaces to remote objects. As such, redefining the method to be an operation would add an exception to the throws clause of the subclass's "method". Java does not permit the addition of exceptions to the throws clause of redefined methods. Support for such a redefinition would require either modifying the language and library such that all methods throw the RemoteException exception or creating a proxy to handle the RemoteException exceptions thrown by operations. Modifying the Java library for this purpose is unacceptable. Creating a proxy for each operation reduces the performance of operation invocations and requires that the proxy code handle the communication exceptions in a general manner. Communication exceptions are currently thrown to the user code (through the RemoteException) so that the programmer may handle them appropriately.

### 5.2.5 Inheritance Anomaly

In concurrent object-oriented languages (COOLs), there exists a conflict between inheritance and synchronization constraints that often requires redefinition of inherited methods in order to maintain the integrity of concurrent objects. This conflict has been deemed the *inheritance anomaly* (see Section 2.5). The complaints regarding the inheritance anomaly are focused on functionally unnecessary redefinition of methods and a violation of encapsulation. The proposals, discussed in [55], that seek to resolve the inheritance anomaly attempt to reduce the amount of redefinition required. Rather than requiring the redefinition of the implementation of a method, many of the proposals allow the redefinition, in some form, of the synchronization constraint. However, the encapsulation argument does not seem to be resolved since the synchronization code in the subclass often takes advantage of knowledge of the synchronization code used in the parent class.

JR inherently suffers from the inheritance anomaly because it allows for the reception of messages within the body of a method, i.e., JR's inni statement appears within executable code. If the synchronization strategy is redefined in a subclass, then some methods may require redefinition in order to change the operations from which they receive. This redefinition is required even if the functionality of the method is to remain the same. The same is true of Java programs that use synchronized blocks (as opposed to synchronizing the entire method) within the bodies of methods (e.g., to emulate semaphores). We feel that the drawbacks of the inheritance anomaly are more than compensated for by the flexibility provided by servicing operations within method bodies and is needed to provide the support argued for in Section 3.5.

## 5.3 Implementation

The JR translator extends the Java compiler available in SUN's JDK, Version 1.2.1. The translator converts JR programs into standard Java programs that are supported by the JR run-time system. The JR run-time system is also implemented in standard Java. This section discusses the interesting parts of the implementation of each of the features provided by JR.

### 5.3.1 JR Virtual Machines

Each JR virtual machine is a small Java program (called `jrvm`) that provides a thin layer over the standard Java virtual machine to allow remote object creation. To create a new virtual machine (using `new` as discussed in Section 5.1.1), a thread must contact a centralized virtual machine manager, called JRX, which plays a role similar to that of SR's SRX [3]. JRX contacts (via `rsh`) the physical host on which the virtual machine is to be created and initiates the execution of the `jrvm` program. Once created, a virtual machine (i.e.g, `jrvm`) uses RMI to contact JRX and register itself as ready to receive requests. A reference to the virtual machine is then returned to the instantiating thread so that remote objects can subsequently be created on the virtual machine.

### 5.3.2 Remote Objects

Remote objects are created, using a `new` expression, on either the local virtual machine or on a remote virtual machine. Upon execution of the `new` expression, the virtual machine on which the object is to be created is contacted (via RMI) and passed the type of the object to be instantiated, the types of each of the constructor's arguments, and each of the arguments. The virtual machine then uses Java's reflection support to create a new instance of the object. A remote reference is then returned to the instantiating thread.

References to remote objects are implemented as `java.io.Serializable` proxy objects containing references to each of the operations defined within the objects class. These references are operation capabilities that also implement the `java.io.Serializable` interface. The subclassing of the remote proxy objects mimics the inheritance hierarchy of the classes with which they are associated. As such, a reference to a remote `java.lang.Object` object can refer to a remote `java.lang.String` object.

### 5.3.3 Operations and Operation Capabilities

Figure 5.14 shows the actual inheritance hierarchy of operation classes in JR. This hierarchy is a specialization of the inheritance hierarchy in Figure 5.1. Each operation

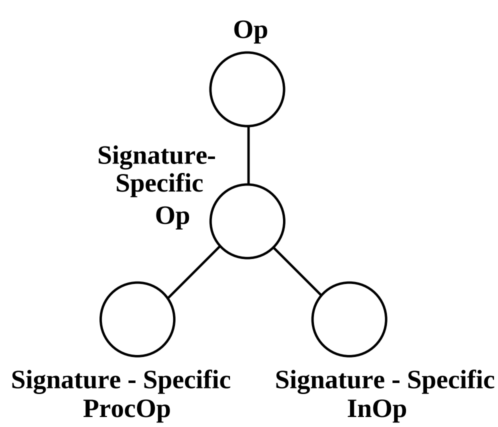



Figure 5.14: Actual JR Operation Inheritance Hierarchy.

serviced by a method is implemented as a separate ProcOp object defined within the class that declares the operation. Thus, a ProcOp object may be serviced by a private method. Invocations of the operation are translated into invocations of the appropriate method (i.e., call, send, etc.) in the ProcOp object. This translation is very similar to the common technique of simulating a method reference (not directly representable in Java without using reflection) as an object with a well-defined interface. Figure 5.15 shows an example of this translation.

```
class A
{
   public op void foo(int);
   public void foo(int i)
   { ... }

   public void bar()
      throws java.rmi.RemoteException
   {
      foo(3);
   }
}
```

(a) JR code.

```
class A
{
   ... // implementation of ProcOp foo

   public void bar()
      throws java.rmi.RemoteException
   {
      JRget_op_foo_intTovoid().call(3);
   }
}
```

(b) Generated Java code.

Figure 5.15: Translation of the invocation of a ProcOp.

An operation serviced by `innis` is implemented as an InOp object that contains a message queue in which to store the arguments for each invocation. An invocation of an InOp is translated into an appropriate method invocation on the corresponding object. All operations implement the `java.rmi.Remote` interface allowing methods to be invoked from remote hosts.

An operation capability is an object that implements the `java.io.Serializable` interface and that contains a reference to the appropriate signature specific operation.

### 5.3.4 Asynchronous Message Passing

Since the JR run-time system is built using RMI, the support for asynchronous message passing is built upon a synchronous method invocation system. As such, the send statement is not truly asynchronous, but is actually semi-synchronous [13]. A send is implemented as an RMI invocation of the send method in the object that corresponds to the operation being invoked. An InOp's send method places a message containing the actual arguments into the invocation queue and then returns. A ProcOp's send method spawns a new thread to execute the method associated with the ProcOp and then returns, releasing the invoking process. This strategy produces the desired effect, but does require the invoker to temporarily block.

### 5.3.5 Inheritance

In Java, a method invocation causes a dynamic lookup to determine the actual method to invoke. However, this lookup is not done when accessing a data field. Since each operation in JR is implemented as a signature-specific Op object, the generated Java code must provide support for dynamic lookup of operations. This support is provided via access methods used to retrieve the appropriate operation object, an example of which can be seen on line 8 of Figure 5.15 (b).

### 5.3.6 Remote Class Loading

The dynamic class loading described in the Java RMI specification [79] allows for class files to be loaded from either the local CLASSPATH or from a predefined URL. The JR run-time system requires only that necessary class files for the program be accessible through the CLASSPATH at the originating host (where the program is initially executed). When a remote object is created, the necessary class files are retrieved from the JRX object on the originating host through a custom network class loader. This reduces the amount of setup required by the user and eliminates the need for a separate server (e.g., an http or ftp server) to provide file access.

## 5.4 Performance Results

A number of microbenchmarks have been used to study the performance of JR programs against equivalent RMI programs. The performance results demonstrate that remote method invocations in JR incur little overhead compared with equivalent invocations in standard RMI (upon which JR is built). Each of the following experiments was conducted on a cluster of 350 MHz Intel Pentium II workstations connected with a 10 Mbps Ethernet network. All experiments were conducted using SUN's JDK, Version 1.3.1 on Linux 2.4.2-2.

The first experiment demonstrates the time needed to invoke an empty method. Table 5.1 shows the results of repeatedly invoking an empty ProcOp in JR and an empty method in Java. The method takes as an argument a single object that contains an array of a specified number of integers.

| Object Size (*ints*) | 1 | $1k$ | $10k$ | $100k$ |
|---|---|---|---|---|
| JR | 0.3682 | 0.3685 | 0.3686 | 0.3694 |
| Java | 0.0741 | 0.0746 | 0.0746 | 0.0758 |
| JR time / Java time | 4.97 | 4.90 | 4.94 | 4.87 |

Table 5.1: Time in microseconds to invoke an empty JR ProcOp and an empty Java method in a local object.

As evidenced by this experiment, a JR method invocation takes about five times longer than a standard method invocation. This is because an invocation of a ProcOp in the current implementation of JR requires four additional method invocations to support dynamic dispatch on operations and causal ordering of messages. The overhead to support dynamic dispatch on operations could be reduced through implementation techniques similar to those used for methods. Further optimization at the compiler level may reduce the overhead of local operation invocations by eliminating the invocations for causal ordering.

The next experiment, taken from [48], extends the previous experiment to measure invocations of an empty method in a remote object. Table 5.2 shows the results of repeatedly invoking an empty ProcOp in JR and an empty method using standard RMI. The remote method takes as an argument a single object that contains an array of a specified number of integers.

| Object Size (*ints*) | 1 | $1k$ | $10k$ | $100k$ |
|---|---|---|---|---|
| JR | 1.73 | 2.28 | 6.64 | 48.43 |
| JDK RMI | 1.60 | 2.19 | 6.21 | 47.63 |
| JR time / RMI time | 1.08 | 1.04 | 1.07 | 1.02 |

Table 5.2: Time in milliseconds to invoke an empty JR ProcOp and an empty RMI method in a remote object.

The performance differences demonstrated in Table 5.2 are attributable to method invocation overhead inherent in the current implementation of JR[3]. A remote method invocation begins by invoking the call method of the operation capability. The operation

[3]The differences are so minor, however, that instances of JR outperforming RMI, due to extraneous network traffic, have been observed.

capability call method invokes the call method of the ProcOp. This invocation transmits the parameters to the remote host using RMI. At the remote host, the ProcOp call method invokes the actual user-defined method.

Table 5.3 shows the results of multiple executions of the Readers/Writers program using both JR and RMI. As Figure 5.5 shows, in the JR solution the Readers/Writers server uses `inni` statements to service invocations of the different InOps associated with requesting and releasing read/write access. The RMI solution in this experiment, however, uses a semaphore-like approach to solving the Readers/Writers problem. Therefore, Table 5.3 includes performance results for a roughly equivalent JR semaphore-like solution.

| R/W/RI/WI | 20/10/3/3 | 50/15/3/3 | 100/30/3/5 |
|---|---|---|---|
| JR (inni) | 3555.0 | 5544.6 | 14428.0 |
| JR (semaphore) | 2765.0 | 4291.6 | 11692.2 |
| JDK RMI | 2723.4 | 3971.6 | 10686.0 |
| JR (inni) / RMI | 1.31 | 1.40 | 1.35 |
| JR (semaphore) / RMI | 1.01 | 1.08 | 1.09 |

Table 5.3: Time in milliseconds to complete execution of all iterations (RI and WI) for all readers (R) and writers (W).

The JR semaphore-like solution uses JR's semaphore abbreviations. These abbreviations translate into sends and receives on an InOp. As such, this semaphore solution does not perform as well as the RMI solution because it guarantees first-come first-serve ordering for threads that execute a `P` operation on a semaphore. The RMI solution provides no such guarantee. Further optimization of the InOp process fairness code should improve the performance of this solution. It should be noted, however, that the JR semaphore solution did outperform the RMI solution when the RMI server was restarted between each test. Restarting the server between tests reduced the effectiveness of Java's Just-In-Time (JIT) compiler for the RMI solution. The results shown in Table 5.3 for the RMI solution are for a persistent server.

The JR solution that uses an `inni` statement did not perform as well as the other two solutions. The performance difference is attributable to the current implementation of the `inni` statement's fairness preserving semantics. Table 5.4 shows the percentage of time spent executing code that pertains to the fairness semantics for the Reader/Writer experiment. As shown in the table, a large percentage of time is spent selecting the invocation to service from the currently pending invocations (e.g., readRequest).

| Functionality | 20/10/3/3 | 50/15/3/3 | 100/30/3/5 |
|---|---|---|---|
| Invocation Sort | 0.3 | 0.7 | 0.7 |
| Select Invocation | 22.2 | 23.9 | 25.5 |
| Remove Invocation | 0.5 | 1.5 | 1.3 |
| Lock Acquire (Wait) | 60.9 | 61.2 | 62.2 |

Table 5.4: JR (inni) Solution: Percentage of total execution time spent executing synchronization code for the Readers/Writers experiment.

Invocation selection takes a large percentage of the total execution time, in this program, because all invocations in an operation are examined until one is found that satisfies the arm's `st` clause. In this program, however, the `st` clause does not reference invocation arguments. Therefore, if the `st` clause is false, all invocations in the operation are examined needlessly. As an optimization for such cases, the `st` clause can be lifted out of the selection loop in the generated code.

Table 5.5 compares the performance results of the standard sequential version of the Java Grande Forum *Fourier* Benchmark [28] against distributed versions written in JR and RMI.

| Number of coefficients | 10000 | 100000 |
|---|---|---|
| Sequential Java | 75.17 | 760.42 |
| JR (1 Server) | 68.01 | 768.37 |
| JR (2 Servers) | 32.29 | 379.85 |
| JDK RMI (1 Server) | 68.63 | 760.00 |
| JDK RMI (2 Servers) | 37.87 | 379.63 |
| JR / RMI (1 Server) | 0.99 | 1.01 |
| JR / RMI (2 Servers) | 0.85 | 1.00 |

Table 5.5: Time in seconds to calculate the first n coefficients of the function $(x+1)^x$ defined on the interval [0,2].

The distributed versions of the program divide the computation equally among the available servers. The JR program uses asynchronous message-passing to initiate each computation and then collects the results using an `inni` statement. The RMI version uses threads to concurrently initiate invocations of the remote method and to collect the results.

## 5.5 Related Work

Other extensions to Java have modified its concurrency model to include, for example, asynchronous communication, distributed shared memory, and active agents. None of these extensions provides the flexibility of operations, capabilities, and `inni` statements. Moreover, many of the previous extensions still require the user to manually start remote programs.

Ajents [48] provides remote object creation, asynchronous RMI, and object migration through a collection of Java classes. The Ajents project makes no modifications to the Java programming language or the run-time system. Creation of remote objects and invocation of methods within remote objects (both synchronous and asynchronous) is done through the `Ajents` class. The arguments to `Ajents.new()`, `Ajents.rmi()`, and `Ajents.armi()` include a String specifying the type of the object to create or the method to invoke. Without a preprocessor, it is not possible to statically determine if an object can be created or if the method being invoked actually exists.

JavaParty [69] extends Java through the addition of the `remote` keyword to provide transparent remote objects. JavaParty includes a translator that converts JavaParty programs into standard Java programs. This translation includes converting the `remote` classes into the corresponding RMI support classes simplifying the work of the programmer.

In [60], Java is extended to include a `remotenew` expression that allows for the instantiation of remote objects on specified hosts. The `remotenew` keyword is mapped to a new opcode which requires an extended virtual machine. The implementation restricts remote method arguments to primitive types whereas JR allows any Serializable or Remote object as an argument.

Asynchronous remote method invocations are provided by [71] by using the `armic` stub/skeleton generator instead of the standard rmic. Unfortunately, all remote method invocations are asynchronous. The programmer is provided access to return values through a mailbox.

A socket implementation of asynchronous message passing is discussed in [40]. This implementation allows a single object to be passed as a message between two threads and provides support for a conditional receive (a very limited `inni` statement). Support for rendezvous is provided via a blocking request at the “client” and a request/reply pair at the “server”.

Both Java/DSM [86] and Charlotte [12] extend Java to include mechanisms for distributed shared memory. Javelin [23], SuperWeb [1], and ParaWeb [16] seek to exploit the potential for parallel computation using the World Wide Web. Communicating Java Threads [41] extends the concurrency model of Java by providing communication between threads based on the CSP [42] paradigm. Support for data-driven objects in Java is discussed in [50].

The current JR implementation uses but does not rely upon RMI. As such, future JR implementations can take advantage of improved or optimized communication frameworks. For example, [53] and [61] discuss more efficient versions of Java's RMI.

Java-SR [57] adds SR operations, asynchronous message passing, and dynamic resource creation to Java. Though a good initial effort, Java-SR has several shortcomings. Java-SR uses a preprocessor to translate programs into standard Java but adds little to the Java syntax. The resulting language is a mix of syntactic extensions to define operations and exposed implementation details for interacting with operations. Java-SR does not truly integrate the SR concurrency model and Java. Java-SR does not support the VM abstraction, lacks support for dynamic remote object creation, and provides a very limited input statement. Furthermore Java-SR does not address the extension of the SR constructs in terms of the Java programming language (e.g., inheritance and redefinition of operations, and exception handling during asynchronous invocation).

More generally, numerous concurrent object-oriented languages have been proposed, e.g., as discussed in a recent survey [18] (discussed in greater detail in Section 2.4). These languages have various concurrency models, ways of expressing synchronization, etc. Some concurrent object-oriented languages take an object-oriented approach to synchronization (as does JR), including Simtalk [14], Actalk [19], GARF [39, 31], and BAST [32, 33]. SimTalk, for example, derives different kinds of monitor classes from a common base class. Although JR shares some features with some concurrent object-oriented languages, JR differs in its overall approach of building synchronization via the operation abstraction, its overall concurrency model, and its definition as an extension of Java.

Furthermore, the survey classifies the different approaches in object-oriented concurrent programming into three coarse categories [18] (see Section 2.4 for more discussion). The three categories are: the *library* approach, the *integrative* approach, and the *reflective* approach. The *library* approach provides class libraries that encapsulate concurrency components (e.g., in Java, threads are represented as objects, see Section 3.1). The *integrative* approach unifies concurrency concepts with object-oriented concepts (e.g., merging the notion of *object* and *process* to create the notion of an *active object*). The *reflective* approach uses reflection mechanisms to provide concurrency components at the meta-level.

JR takes an *integrative* approach in defining its concurrency model. JR provides operations as a general communication abstraction. Operations are defined as part of an object and each operation is associated with a specific object. Using operations, a JR programmer can create active objects, synchronized objects, and distributed objects. An object can be made active by using the `process` keyword to create a thread within the object. A JR programmer can use Java's `synchronized` keyword or InOps to synchronize invocations. As discussed in Section 5.1.2, JR provides dynamic remote object creation and JRobject capabilities to facilitate the distribution of objects.

## 5.6 Summary

The JR programming language integrates the SR concurrency model and the Java programming language. It does so via a novel approach that resolves the tension between inheritance and concurrency. JR provides a more flexible way to program distributed applications without great performance costs. An additional outcome of the integration of the SR concurrency model with Java is support for exception-handling during asynchronous method invocation. This topic is addressed separately in Chapter 6.

In addition to the object-oriented features discussed in this chapter (which SR does not provide), JR supports all of the concurrent programming features of SR with the exception of the concurrent invocation statement, quantifiers in input statements (typically used to specify indices of operation arrays), and termination detection. The concurrent invocation statement and quantifiers on the `inni` statement could, with reasonable effort, be added to JR but have not been added at this point. Some initial research, however, indicates that it may not be possible to add termination detection to JR without modifying the Java virtual machine (something that this work has thus far avoided in the interest of portability).

# Chapter 6

# Exception Handling During Asynchronous Method Invocation

Asynchronous method invocation facilitates the dynamic creation of concurrently executing threads and the communication between such threads. When a method is asynchronously invoked, the invoking thread continues to execute while another thread executes the body of the invoked method. Such concurrent execution can benefit both the design of and the performance of a program. The use of asynchronous method invocation, however, complicates the use of exceptions and, in particular, the handling of thrown exceptions.

Exception handling mechanisms for sequential programming languages are well-understood. Goodenough [37] presents a general discussion of the issues that exception handling mechanisms need to address and the different semantics (e.g., terminate, retry, or resume) provided by such mechanisms. Yemini and Berry [85] propose an additional model (replacement) that allows an erroneous subexpression (one that raises an exception) to be *replaced* by the handler's result.

The exception handling mechanisms provided by sequential programming languages rely upon the call stack for the propagation of exceptions. A thrown exception is propagated (either implicitly or explicitly) up the call stack until an appropriate handler is found. Figure 6.1 depicts such a propagation. Method `baz` throws an exception. The exception is propagated through method `bar` and into method `foo`, where it is finally handled.

Unfortunately, the reliance of such mechanisms on the call stack is inadequate for

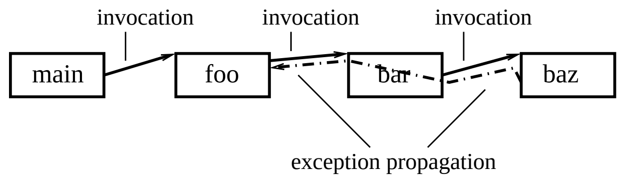



Figure 6.1: Exception propagated through call stack.

handling exceptions thrown from an asynchronously invoked method. Figure 6.2 depicts the same program as before, but with the method `bar` invoked asynchronously. Again, an exception is thrown from method `baz` and propagated to method `bar`. If method `bar` does not have an appropriate exception handler, then the exception must be further propagated. But, since method `bar` was invoked asynchronously, the preceding call stack is not accessible. In fact, the method that invoked `bar` (in this example, `foo`) may no longer be executing. Therefore, the call stack cannot be used to propagate exceptions from methods invoked asynchronously.

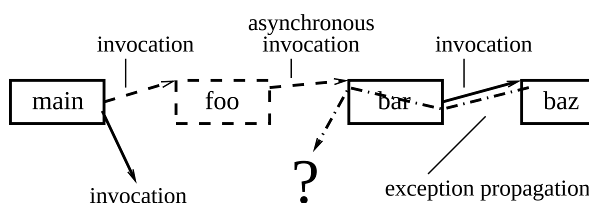



Figure 6.2: Exception propagated from method invoked asynchronously.

This problem is addressed, in JR, by requiring the specification of handlers for exceptions thrown from methods invoked asynchronously. This approach is similar to the approaches taken in both ABCL/1 [44] and Arche [47], which are discussed in greater detail in Section 6.6. JR's approach, however, differs in the support for static checks for handling exceptions, handling exceptions after an early reply from a method, handling exceptions after forwarding within a method, and handling exceptions with operations (a generalization of a method). Both early reply and forwarding are useful in distributed programming ([52] and [6]). Early reply allows the invoked method to transmit return values to its invoker, yet the invoked method continues to exist and executes concurrently with its invoker. Early reply facilitates programming conversational continuity. Forwarding passes the responsibility for replying to the invoker from the invoked method to another method. Forwarding facilitates partial processing of an invocation, so that, for example, a manager process can use the contents of an invocation to decide which of several worker processes should be assigned the request.

The exception handling support discussed in this chapter has been designed and implemented as part of the JR programming language[1]. As such, all examples in this chapter use JR notation. This exception handling model, however, should be useful in other concurrent languages and systems that support exceptions and asynchronous invocation. For example, CORBA [62] supports *one-way* invocations but does not allow methods invoked in such a manner to throw exceptions.

The rest of this chapter is organized as follows. Section 6.1 presents the basic model used to handle exceptions during asynchronous method invocation. Section 6.2 extends the basic model to provide support for handling exceptions thrown from a method after an early reply. Section 6.3 further extends the model to provide support for handling exceptions thrown from a method after forwarding. Section 6.4 discusses the potential for using operations (a generalization of a method) to handle the exceptions thrown during asynchronous invocations. Section 6.5 gives an overview of the implementation of the exception handling model. Section 6.6 discusses related work. Finally, Section 6.7 summarizes this chapter.

## 6.1 Exceptions During Asynchronous Invocation

Recall from Section 5.1.5 that the JR programming language extends Java with support for, among other features, asynchronous method invocation via the `send` statement. Execution of a send statement will, as expected, create a thread to execute the invoked method. To facilitate the handling of exceptions thrown from an asynchronously invoked method, JR requires the specification of a *handler* object as part of a `send`. Any exceptions propagated (thrown) out of the invoked method will be directed to the *handler* object. As such, in accordance with the verbose nature of Java's exception handling model, JR requires that the specified *handler* object is capable of handling any exception potentially thrown

[1]In the interest of generality, the ensuing discussion refers only to methods. The model also supports handling exceptions thrown from operations and, as a result, input statements in JR.

from the invoked method[2]. This requirement is statically checked (see Section 6.1.2).

### 6.1.1 Handler Objects

To be used as a *handler*, an object must implement the `Handler` interface and define a set of *handler* methods. A method is defined as a *handler* through the use of the `handler` modifier (much like the use of the `public` modifier)[3]. A *handler* method can take only a single argument: a reference to an exception object. Each *handler* method specifies the exact exception type that it can handle. When an exception is delivered to a *handler* object, it is handled by a *handler* method of the appropriate type. If there are two handler methods that may handle the exception, then the method with the most specific type is selected. *Handler* methods are not allowed to propagate exceptions out of the method body.

An example definition of a *handler* object's class is given in Figure 6.3. In this example, *handler* objects of type `IOHandler` can handle end-of-file and file-not-found exceptions. An exception of type `java.io.EOFException` directed to such a *handler* object will be handled by the `handleEOF` method (and similarly for `java.io.FileNotFound` exceptions).

```
public class IOHandler
    implements edu.ucdavis.jr.Handler
{
    public handler void handleEOF(java.io.EOFException e)
    { /* handle exception */ }

    public handler void handleNotFound(java.io.FileNotFoundException e)
    { /* handle exception */ }
}
```

Figure 6.3: Class definition for a simple *handler* object.

[2]Java requires that a method specify, using a `throws` clause, all exceptions that may potentially propagate out of the method.

[3]JR uses a keyword to specify *handler* methods. Another option is to impose a naming convention on *handler* methods, but doing so would necessarily restrict the naming of methods.

### 6.1.2 Send

A `send` statement must specify, using a `handler` clause, the *handler* object that is to be used to handle any exceptions propagated from the asynchronously invoked method[4]. The *handler* object must be capable of handling any exceptions thrown from the method. Such an object, defined as discussed above, must define a *handler* method for each exception thrown[5].

An example of the specification of a handler object is given in Figure 6.4. The JR compiler statically checks that the specified *handler* object can handle each of the potentially thrown exceptions.

```
IOHandler iohandler = new IOHandler();
...
send readFile("/etc/passwd") handler iohandler;
...
```

Figure 6.4: Specification of a *handler* object during a send.

## 6.2 Exceptions after Early Reply

When synchronously invoking a method, the invoker must wait (block) until the invoked method finishes executing (i.e., returns). In languages supporting early reply (see Section 4.4), however, an invoker need only wait until the method signals the invoker to continue, after which both the invoker and the invoked method execute concurrently. Figure 6.5 demonstrates the use of a `reply` statement within method `foo` to return the computed value of `retval` to the invoker, method `bar`.

Much like an asynchronous invocation, an early reply breaks the call stack link between the invoking method and the invoked method. As such, exceptions thrown after a reply cannot be propagated back to the invoking method. To address this problem, a `reply` statement must specify a *handler* object that is capable of handling any exceptions potentially thrown by the replying method, for example, as shown in Figure 6.5. Exceptions thrown prior to the execution of a `reply` statement are propagated as appropriate

[4]If the operation cannot propagate exceptions, then a *handler* object need not be specified.

[5]More precisely, the *handler* methods may handle exceptions that are supertypes of the thrown exceptions, resulting in potentially fewer *handler* methods than exceptions.

```
int foo(String filename) throws java.io.EOFException
{
  int retval = 0;
  ... // calculate retval
  reply retval handler iohandler; // early reply
  ... // continue executing
}

void bar(String filename)
{
  int value = foo(filename);
  // execute concurrently with foo after foo replies
  ...
}
```

Figure 6.5: Early reply.

for the manner in which the method was invoked (i.e., through the call stack if invoked synchronously and to a *handler* object if invoked asynchronously). Exceptions thrown after a reply are directed to the *handler* object specified as part of the reply.

## 6.3 Exceptions after Forwarding

Another modification of the standard synchronous invocation semantics is forwarding (see Section 4.4). An invoked method may *forward* to another method the responsibility of replying. Figure 6.6 gives an example of forwarding. In this example, method `foo` does some calculations and then forwards responsibility to method `bar`, after which the two methods execute concurrently.

As with a `reply` statement, a `forward` statement must specify a *handler* object to handle any exceptions thrown after forwarding. Again, any exceptions thrown prior to executing a `forward` statement are propagated according the the manner in which the method was invoked. Any exceptions thrown after a `forward` statement are directed to the *handler* object.

A key difference with the `forward` statement is that the method to which responsibility is forwarded inherits the call stack link from the forwarding method. So, in the example in Figure 6.6, method `foo` was synchronously invoked and forwarded the responsibility for replying to the invocation to method `bar`. After forwarding, the call stack link

```
int baz(String filename) throws java.io.EOFException
{
  int retval = foo(filename);
  // retval actually comes from bar because of foo's forward
  ...
}

int foo(String filename) throws java.io.EOFException
{
  ...
  IOHandler iohandler = new IOHandler();
  forward bar(filename) handler iohandler; // forward invocation
  ... // continue executing
}

int bar(String filename) throws java.io.EOFException
{
  ...
}
```

Figure 6.6: Forwarding.

that had been between `baz` and `foo` is changed to link `baz` and `bar`. As such, exceptions thrown from method `bar` will be propagated through the call stack back to method `baz` (the original invoker of method `foo`). Similarly, if method `foo` had been invoked asynchronously, any exceptions thrown from method `bar` would be directed to the *handler* object specified as part of the asynchronous invocation of method `foo`.

## 6.4 Exceptions and Operations

Section 6.1.1 discusses the definition of *handler* objects. Each *handler* object must define *handler* methods to actually handle exceptions. These methods may actually be operations. Recall from Section 5.1.3 that an operation is a generalization of a method. An operation may be invoked either synchronously or asynchronously and may be implemented by either a method or by an input statement (a multi-way receive). When an operation implemented by a method is invoked, the method is executed. When an operation implemented by an input statement is invoked, a message is placed into a queue until another thread executes an input statement to receive the message. Once received, the input statement executes a body of code to service the message.

The use of operations provides greater flexibility in handling exceptions. In particular, the use of operations serviced by input statements allows the handling of exceptions to be distributed among a number of threads (each executing an input statement servicing the operation) or serialized by a single thread servicing multiple operations with a single input statement. Distributing exception handling allows exceptions to be routed to the appropriate components of the system. Serializing exception handling can provide a means for prioritizing exception handling.

Figure 6.7 shows the class definition of a *handler* object that handles exceptions using an input statement (`inni`). The `server` thread (declared using the `process` notation) repeatedly services exceptions (one at a time) as they are raised to the *handler* object.

```
public class IOHandler
   implements edu.ucdavis.jr.Handler
{
   public handler op void handleEOF(java.io.EOFException e);

   public handler op void handleNotFound(java.io.FileNotFoundException e);

   process server
   {
      while (true)
      {
         inni void handleEOF(java.io.EOFException e)
            { /* handle java.io.EOFException exception */ }
         [] void handleNotFound(java.io.FileNotFoundException e)
            { /* handle java.io.FileNotFoundException exception */ }
      }
   }
}
```

Figure 6.7: A *handler* object that uses operations.

## 6.5 Implementation

*Handler* objects are implemented as *Serializable* Java objects. The *handler* methods are implemented as normal methods, but are gathered during compilation to generate a dispatch method. The dispatch method is necessary as a single, well-named entry point into each *handler* object. All exceptions are directed to the dispatch method, which then routes the exceptions to the appropriate *handler* methods. A *handler* object is sent, as an additional parameter, to an invoked method, and used only when an exception is raised. Since the *handler* object is sent as part of the invocation, it will exist during the duration of the method execution[6].

To support detaching an invoked method from an invoking method (i.e., breaking the call stack link), any method that can potentially execute a `reply` or `forward` statement is modified to return control (reply) to its invoker through a message channel. The invoking method waits to receive a reply from the invoked method through this message channel. This message channel acts as the call stack link between invoker and invokee, and it is this channel that is inherited by a method that has been forwarded to. If the message channel does not exist, then the method is detached. This indicates that the method was invoked asynchronously, the method has replied, or the method has forwarded.

If a method is not detached, then exceptions are propagated through the call stack (using message channel links where appropriate). When an exception is propagated from a detached method, the exception is sent to the *handler* object supplied for the method. The *handler* object then routes the exception to its appropriate *handler* method.

## 6.6 Related Work

There has been previous research on exception handling in conjunction with asynchronous method invocation. For example, [81] allows specification of the exception handler when the exception is actually raised. As proposed, the exception handler is another process (which must be interrupted), but the idea is easily extensible to allowing objects as handlers. Unfortunately, specifying the handler at the point an exception is raised causes some limitations. If the method can be invoked synchronously, then it might be desirable to propagate the exception up the call stack rather than to the handler. Allowing such a distinction would require support for a method to determine the manner in which it has been invoked, otherwise the exception will always be raised to either another process/object or always propagated through the call stack.

[6]It is possible, using an explicit cast, to pass a `null` reference as a handler object. Attempts to raise an exception to a `null` reference, or to a remote object that no longer exists (e.g., due to network partitioning or a host failure), will be caught and cause the program to halt.

As mentioned previously, the solution proposed in this chapter bears some resemblance to the solutions for ABCL/1 [44] and Arche [47]. ABCL/1 allows synchronous, asynchronous, and future-based method invocation. Each invocation may specify a "complaint" destination. If a "complaint" destination is specified, then any exception raised during the execution of the method will be directed to the complaint object. Otherwise, the exception is propagated to the invoker (or an alternate "reply" destination, if specified), again relying upon the call stack. ABCL/1, due to the nature of the language, does not perform static checks on the "complaint" destination to ensure that the destination can handle the potentially thrown exceptions.

The exception handling support in Arche is similar to that provided by ABCL/1. A set of handler objects can be specified as part of an asynchronous invocation. Exceptions thrown from the invoked method are directed to each of the specified handler objects. Arche also checks that each of the handler objects can actually handle the potentially thrown exceptions.

JR's support differs from the support provided by both ABCL/1 and Arche in that it also addresses exception handling after early reply, after forwarding the responsibility to reply, and via operations.

The design of an exception handling mechanism for the SR concurrent programming language (from which JR's concurrency model is derived) is discussed in [43]. This work takes a novel approach in the design of its exception model through the use of operations (SR's communication primitive). The work leverages the flexibility of operations to provide modern exception handling capabilities. The exception model, however, was not actually implemented and, since SR lacks any support for exception handling, implementing the exception model would be non-trivial. Furthermore, the work does not directly address the handling of exceptions raised during asynchronous invocation. Nonetheless, since the exception handling model uses operations, it is, in theory, possible for a programmer to explicitly design a handling strategy akin to handler objects. Doing so, however, is likely to be necessarily complicated.

Recent work [54] has extended the functional language Haskell to support handling exceptions raised by events external to a program (e.g., timeouts). The authors of [54] call such exceptions asynchronous. Unlike the exceptions addressed in this paper, their asynchronous exceptions are not, necessarily, raised as a result of a method invocation[7]. Their asynchronous exceptions can, however, be thrown from one thread to another, and, as such, do bear some relation to the exceptions discussed in this paper. In their model, a thread explicitly blocks exception delivery until an exception handler has been established. This handler can catch either synchronous exceptions (thrown as a result of evaluating the expressions within the try block) or asynchronous exceptions (delivered during the evaluation of the expressions within the try block). This strategy relies on the existence of the receiving thread (and considers an exception thrown to a dead or completed thread trivially handled), which we want to avoid.

## 6.7 Summary

This chapter presented the design and implementation of an exception model that supports the handling exceptions thrown from an asynchronously invoked method, handling exceptions thrown after an early reply from a method, and handling exceptions after forwarding the responsibility to reply. Furthermore, the chapter discussed the use of operations for handling exceptions.

The presented exception model uses *handler* objects to handle exceptions that cannot be propagated up the call stack. These *handler* objects are carried with the parameters as part of an invocation. As such, this exception model can be used in other languages and systems that support asynchronous invocation.

[7] It is because of this distinction that we do not refer to the exceptions we are addressing as asynchronous.

# Chapter 7

# Inter-operation Invocation Selection Mechanism

Many concurrent programming languages support message passing as a form of thread synchronization (Section 2.3.4). A message, henceforth called an *invocation*, is generated when a thread invokes an operation (see Section 5.1.3), optionally passing it parameters. Once generated, the invocation is added to the set of pending invocations associated with the invoked operation. An invocation is removed from an operation's set by a servicing thread using an invocation servicing mechanism. Such a mechanism, depending on the language, may allow selection of an invocation from a single operation (a receive) or from one of potentially many operations (a multi-way receive).

This work focuses on support for multi-way receives and the invocation selection support provided. Ada [45], Concurrent C [35], CSP [42], Hermes [77], JR [51], Limbo [27], occam [21], Orca [9], SR [4, 3], and SRR [24] each provide support for multi-way receives, but with differing expressive power. None of these languages provides general support for inter-operation invocation selection. Specifically, these languages do not provide simultaneous access to all pending invocations (both within an operation's set and over the set of operations) during selection. Inter-operation selection facilitates the implementation of, for example, lottery scheduling for resource management (in which a manager thread picks a request randomly) [83] and preferential job scheduling (in which preference is given to normal interactive jobs unless there is a superuser batch job) [24].

This chapter presents a study in the design and analysis of four proposed inter-operation invocation selection mechanisms that address the limitations discussed above. These mechanisms facilitate the implementation of an important class of selection algorithms: those that require access to all pending invocations (both within an operation's queue and over the set of operations). The designs combine aspects of functional, object-oriented, and concurrent programming languages. The result of this study is the addition of the most balanced (in terms of programmer-level abstraction and performance) mechanism to the JR concurrent programming language [51].

The rest of this chapter is organized as follows. Section 7.1 discusses the limitations of multi-way receives provided by other languages. Section 7.2 describes the details of the proposed selection mechanisms. Section 7.3 analyzes, both qualitatively and quantitatively, the proposed mechanisms. Section 7.4 compares the proposed mechanisms and provides justification for the mechanism added to JR. Finally, Section 7.5 summarizes the results of this chapter.

## 7.1 Background

Many concurrent languages provide support for multi-way receives, but with limited expressive power. This section presents representative examples of the multi-way receives supported by different concurrent languages and addresses their support for invocation selection.

### 7.1.1 Ada

Ada provides the `select` statement, which allows multiple `accept` statements, each servicing an operation. When a `select` statement is executed, an operation is non-deterministically selected from those with pending invocations, and an invocation from the selected operation is serviced. Figure 7.1 demonstrates, using JR syntax,

a simple two-way receive that services invocations from two operations. Each `accept` statement may be preceded by a boolean guard to restrict the operations considered for servicing. A guard, however, cannot access an invocation's arguments and, thus, cannot use the values of these arguments in deciding which invocation to select. These guards provide a very coarse-grained control over invocation selection.

```
1   public class Server
2   {
3      public op void entryOne(int priority, int b);
4      public op void entryTwo(int priority, float b);
5
6      protected void server()
7      {
8         /* repeatedly service an entryOne invocation or an entryTwo invocation */
9         while (true)
10        {
11           inni void entryOne(int priority, int b)
12              { /* service */ }
13           [] void entryTwo(int priority, float b)
14              { /* service */ }
15        }
16     }
17  }
```

Figure 7.1: Simple Two-way Receive.

### 7.1.2 SR

SR provides a more flexible invocation selection statement called the input (`in`) statement (see Chapter 4). A guard on an arm of an input statement can contain a *synchronization* expression and a *scheduling* expression (the latter of which Ada does not provide). The former specifies which invocations are acceptable for servicing; the latter specifies the order in which to service acceptable invocations. Figure 7.2 shows a modification of the two-way receive from the earlier example, lines 11–14 in Figure 7.1. This modified two-way receive uses scheduling expressions (specified via `by` clauses) to service each operation's invocations in order of highest priority (lowest integer value). Unlike the guards in Ada, SR's synchronization and scheduling expressions can access an invocation's arguments and use their values to determine an invocation's acceptability and to order invocations. Such expressions, however, cannot simultaneously access the arguments of multiple invocations, and cannot be used to order invocations between multiple operations. As such, in SR, invocation selection considers only a single invocation at a time.

```
inni void entryOne(int priority, int b) by -priority
    { /* service */ }
[] void entryTwo(int priority, float b) by -priority
    { /* service */ }
```

Figure 7.2: Two-way Receive with Scheduling Expression.

### 7.1.3 SRR

SRR extends SR to provide the `rd` statement, which allows the examination of all pending invocations of a set of operations. The `rd` statement, however, cannot directly service an invocation. An invocation is marked for service by a `mark` statement and is serviced by a subsequent `take` statement. Though this approach allows invocation selection based on the set of pending invocations, the separation of the invocation selection into three non-atomic stages (i.e., `rd`, `mark`, and `take`) can complicate solutions. For example, an `rd` statement can be used to find and `mark` the highest priority invocation among a set of operations. Without atomicity, however, the `take` statement to service the invocation may fail because the invocation may have been serviced by another thread.

### 7.1.4 Other Approaches

Other approaches (e.g., [29, 30, 21, 56]) seek to increase selection flexibility through the specification of preferences for the operations being serviced. Preferences define (either statically or dynamically) a partial ordering of the operations serviced by a multi-way receive. This ordering dictates only the order in which operations are checked for serviceable invocations. Preferences do not provide access to the arguments of the pending invocations of an operation or of a set of operations.

### 7.1.5 Desired Support

None of the aforementioned mechanisms provides support for implementing selection algorithms that require atomic access to the entire set of pending invocations. This class of selection algorithms includes debugging, visualization, and scheduling. Extending the example in Figure 7.2, these mechanisms cannot enforce the priority ordering among operations[1]. As such, an invocation in `entryTwo` may be serviced even if `entryOne` has a higher priority invocation pending. Figure 7.3 gives pseudo-code for a solution to the priority selection problem that hints at the type of inter-operation invocation selection mechanism desired.

```
inni "select highest priority invocation from operations"
   void entryOne(int priority, int b)
   { /* service */ }
[] void entryTwo(int priority, float b)
   { /* service */ }
```

Figure 7.3: (pseudo-code) Two-way Receive with Inter-operation Selection.

As another example, consider servicing user requests according to a lottery scheduling [83] algorithm in which requests are randomly selected (according to user id) and serviced for a fixed quantum. This selection algorithm cannot be codified using the aforementioned mechanisms because there is no facility by which to express randomness. The desired support is, again, hinted at by the pseudo-code in Figure 7.4.

```
inni "select an invocation according to the randomly determined user id"
   void entryOne(int a, int b)
   { /* service */ }
[] void entryTwo(int a, float b)
   { /* service */ }
```

Figure 7.4: (pseudo-code) Two-way Receive with Lottery Scheduling.

### 7.1.6 General Approach

Existing invocation selection mechanisms provide built-in support for very specific selection strategies. For example, the synchronization expressions of SR and guards of Ada allow a sort of filtering that can eliminate invocations for consideration for servicing. Such filtering is useful for many selection algorithms, but this built-in support is inflexible and complicates the implementation of selection algorithms that do not fit the filtering mold.

[1]Such an ordering can be provided as discussed in [65], but this requires a change to the interface of the operations.

The approach taken in designing the invocation selection mechanisms presented in this chapter is to balance complexity with flexibility. These mechanisms provide a framework on which the programmer can build (by writing some additional specific selection code) to implement a wide range of selection algorithms.

## 7.2 Proposed Invocation Selection Mechanisms

An acceptable inter-operation invocation selection mechanism must:

- allow for the selection of any pending invocation in the set of operations being serviced.
- provide access to each invocation's actual arguments.
- disallow the removal of multiple invocations from the set of pending invocations .
- disallow the insertion of an invocation into the set of pending invocations as a side-effect of selection.

The first two criteria (selection and argument access) address the functionality desired for the proposed invocation selection mechanisms. The latter two criteria address safety concerns consistent with the semantics of the input statement in JR: only the single invocation that is to be serviced is removed from the set of pending invocations[2]. Additionally, all selection is atomic. A thread executing an `inni` statement locks all of the operation queues until an invocation has been selected (and removed) for servicing or the selection fails (and the thread blocks).

This chapter studies three approaches that satisfy these criteria: Invocation Enumeration, Functional Reduction, and Hybrid. The Invocation Enumeration approach provides an enumeration[3] of the pending invocations. This enumeration is passed to a selection method that determines which invocation to service. The Functional Reduction approach allows the programmer to specify methods that select a representative invocation for each operation. Another function is then used to select the invocation to service from the set of representative invocations. The Hybrid approach combines aspects of the other approaches. Representative invocations are selected by methods and then passed, via an enumeration,

[2] A further extension could relax these constraints to allow several invocations to be simultaneously serviced.

[3] An enumeration (similar to an iterator) generates a series of elements, one at a time (but disallows inserts and deletes).

to a selection method that returns the invocation to service. Each of these approaches is discussed in detail below.

### 7.2.1 Invocation Enumeration

The Invocation Enumeration approach provides an enumeration of the currently pending invocations. This enumeration is passed to a programmer-specified method that selects an individual invocation. The `inni` statement services the invocation that this method returns.

This chapter introduces two variants of Invocation Enumeration: View Enumeration and Named Enumeration. These variants differ in how they name invocations and invocation parameters. View Enumeration names each invocation's arguments as invocations are extracted from an enumeration. Named Enumeration names invocations as they are placed into an enumeration. As will be discussed in Section 7.4, View Enumeration sacrifices reusability for performance whereas Named Enumeration does just the opposite.

```
1   public op void entryOne(int i, float f);
2   public op void entryTwo(int i, float f, float g);
3
4   Invocation select_method(ArmEnumeration arm)
5   { ... }
6
7   ...
8   inni with select_method over
9      void entryOne(int i, float f)
10     { ... }
11  [] void entryTwo(int i, float f, float g)
12     { ... }
13  ...
```

Figure 7.5: Invocation Enumeration: Specification of selection method.

Both approaches use a `with-over` clause to specify the selection method to which the invocation enumeration is passed (demonstrated on line 8 of Figure 7.5). The method must have type signature $ArmEnumeration \rightarrow Invocation$[4]. An `ArmEnumeration` enumerates `InvocationEnumerations`, each corresponding to the operation serviced by an arm of the `inni` statement. Each `InvocationEnumeration` is an enumeration of the respective op-

[4] `ArmEnumeration`, `InvocationEnumeration` and `Invocation` are predefined classes for these approaches.

eration's pending invocations. Figure 7.6 gives a pictorial example of an arm enumeration that enumerates invocation enumerations, each of which, in turn, enumerate invocations.

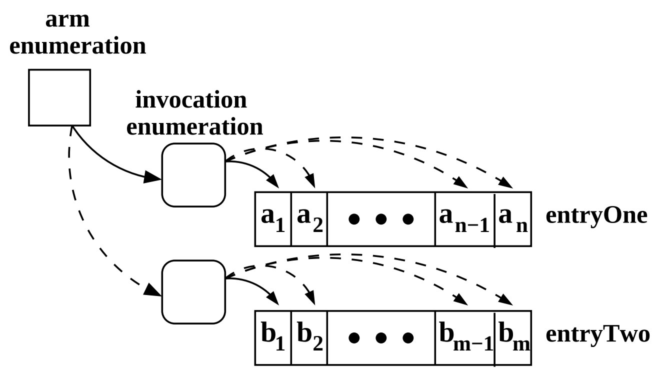



Figure 7.6: Arm and Invocation Enumeration Structure.

#### 7.2.1.1 View Enumeration

An abridged implementation of the selection method used in Figure 7.5 is given in Figure 7.7. This example demonstrates accesses to individual invocations and their arguments. Line 7 shows the extraction of an `InvocationEnumeration` from the `ArmEnumeration`. An `Invocation` is extracted from an `InvocationEnumeration` on line 11. Under View Enumeration, the class `Invocation` is viewed as a variant type [22] of all invocation types and each invocation as a value of this variant type. A `view` statement is used to determine the underlying type of a specific invocation and to provide access to the arguments of the invocation[5]. The `view` statement on lines 12–15 compares invocation `invoc` to the underlying invocation types $int \times float$ and $int \times float \times float$. The statement associated with the matching `as` clause is executed with the invocation's arguments bound to the respective parameters.

#### 7.2.1.2 Named Enumeration

The View Enumeration approach requires that the structure of each `view` statement closely match that of the associated `inni` statement. This structural dependence potentially limits selection method reuse. The Named Enumeration approach addresses this coupling problem by reducing the structural dependence of the selection method on

[5] The standard *case* syntax is not used to avoid confusion with Java's use of *case*.

```
Invocation select_method(ArmEnumeration arm)
{
  Invocation ret_invoc = null;
  ...
  while (arm.hasMoreElements())
  {
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum == null) continue;
    while (invoc_enum.hasMoreElements())
    {
      Invocation invoc = invoc_enum.nextElement();
      view invoc as (int i, float f)
        ... ret_invoc = ... // statement using i and f
      as (int i, float f, float g)
        ... ret_invoc = ... // statement using i, f, and g
    }
  }
  ...
  return ret_invoc;
}
```

Figure 7.7: Invocation Enumeration: Implementation of selection method.

the `inni` statement. This is done by providing the programmer with a means to specify the invocation type as seen by the selection method. This slight decoupling can increase the reuse potential of the selection method.

Named Enumeration allows the programmer to "name" the object type used within an invocation enumeration. These types are specified via `as` clauses as demonstrated on lines 9 and 11 of Figure 7.8. These object types, `nameOne` and `nameTwo` in the example, must extend the `Invocation` class and provide a constructor with type signature matching that of the operation being serviced (excluding, of course, the return type). The definition of `nameOne` (with an appropriate constructor) is given in Figure 7.9 along with a demonstration of its use when accessing invocations and their arguments (lines 16 and 17) within a selection method. As will be demonstrated in Section 7.3.4, the creation of these "named" objects can adversely affect performance.

```
public op void entryOne(int i, float f);
public op void entryTwo(int i, float f, float g);

Invocation select_method(ArmEnumeration arm)
{ ... }

...
inni with select_method over
   void entryOne(int i, float f) as nameOne
   { ... }
[] void entryTwo(int i, float f, float g) as nameTwo
   { ... }
...
```

Figure 7.8: Named Enumeration: Specification of selection method and invocation type names.

```
class nameOne extends Invocation
{
   public int argumentOne; public float argumentTwo;

   public nameOne(int _i, float _f)
   {
      this.argumentOne = _i;
      this.argumentTwo = _f;
   }
}

Invocation select_method(ArmEnumeration arm)
{
   ...
      Invocation invoc = invoc_enum.nextElement();
      if (invoc instanceof nameOne)
      ... = ((nameOne)invoc).argumentOne;
   ...
}
```

Figure 7.9: Named Enumeration: Implementation of selection method.

#### 7.2.1.3 Satisfaction of Acceptability Criteria

Each Invocation Enumeration approach provides access to pending invocations through an enumeration and access to the respective invocation arguments through references to the individual invocations. Only the single invocation returned from the reduction method is removed from the set of pending invocations and serviced. Moreover, since direct access to the underlying implementation is not provided, it is not possible to remove or insert additional invocations.

### 7.2.2 Functional Reduction

The Functional Reduction approach splits invocation selection into two phases to provide a simple means for accessing the invocation parameters. Figure 7.10 gives a pictorial example of the two phases. The first phase (depicted by the horizontal arrows), which has direct access to the parameters of the invocations, selects a single invocation from each arm of the `inni` statement; this invocation is called the *representative invocation* for its respective arm. The second phase (depicted by the vertical arrow), which can access invocation parameters through user-defined invocation types, selects the actual invocation to service from the set of representative invocations. Each phase selects an invocation through a reduction, using a programmer-specified method, over the appropriate set of invocations.

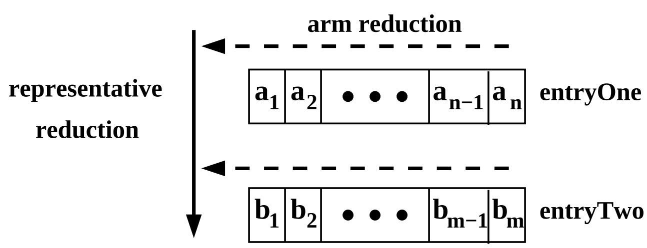



Figure 7.10: Arm Reduction and Representative Reduction Structure.

The notion of a reduction over a set of values is borrowed from functional programming. In general, the higher-order function `reduce` takes a function, `F`, and a list, `L`, and returns the result of applying `F` to the first element of `L` and the result of the reduction of the remainder of `L`. For example, `reduce`, as defined in ML in [82], is

```
fun reduce (F, nil) = raise EmptyList (* error *)
| reduce (F, [a]) = a
| reduce (F, x::xs) = F(x, reduce(F, xs));
```

This type of reduction[6] is used in Functional Reduction. As such, the programmer-specified

[6]Technically, this reduction (as well as that used in the Hybrid approach) is more akin to *foldl*[82] with

```
1   public op void foo(int i, String s);
2
3   Invocation select_arm_method(Invocation prev, Invocation cur, int i, String s)
4   {
5      // do select between cur and prev invocations
6   }
7
8   ...
9   inni void foo(int i, String s) asb select_arm_method
10  { ... }
11  ...
```

Figure 7.11: Functional Reduction: Specification of arm reduction method.

method is written to select between two invocations. Later examples will illustrate the use of this approach to select a single invocation based on information from all pending invocations. Functional Reduction provides an elegant solution to the inter-operation selection problem. The following sections, however, will show that solutions using this approach can become unwieldy.

#### 7.2.2.1 Reduction over a Single Arm

The first phase of Functional Reduction reduces over each individual arm. Reduction over a single arm is specified by an `asb`, "as selected by", clause, which denotes the method to use in the reduction. An example of such a clause is shown on line 9 in Figure 7.11. Only methods that satisfy the typing constraints of the `asb` clause can be used. The typing constraint requires that if the operation (`foo` in Figure 7.11) has type signature $(\alpha_1 \times \cdots \times \alpha_n) \rightarrow \beta$ , then the type signature of the arm reduction method must be $(Invocation' \times Invocation \times \alpha_1 \times \cdots \times \alpha_n) \rightarrow Invocation'$. The type of the first argument must match the return type because the value returned from the $n^{th}$ call of the reduction method is passed as the first argument to the $(n+1)^{th}$ call of the reduction method. $Invocation'$ may be a subclass of `Invocation`, allowing the programmer to specify specialized invocation types that store state to assist in invocation selection. In the example given, `foo` has type signature $(int \times String) \rightarrow void$, so the arm reduction method must have signature $(Invocation' \times Invocation \times int \times String) \rightarrow Invocation'$ (which `select_arm_method`

*null* as the base value. As such, invocations are processed in order of longest pending invocation within each arm.

does, where $Invocation'$ equals $Invocation$).

The typing constraint is required to guarantee that the arm reduction method can take, as arguments, a reference to the "previous" invocation (the result of the previous call to the reduction method), a reference to the "current" invocation (the invocation at the current head of the list being reduced), and each argument of the "current" invocation, and to guarantee that the method will return a reference to an invocation. The individual arguments of an invocation are passed to the arm reduction method, because the arguments are not accessible through an `Invocation` reference [7].

```
1   public op void entryOne(int i, float f);
2   public op void entryTwo(int i, float f, float g);
3
4   Invocation select_method(Invocation prev, Invocation cur)
5   {
6     // do select between cur and prev invocations
7   }
8
9   ...
10  inni with select_method over
11     void entryOne(int i, float f) asb ...
12     { ... }
13  [] void entryTwo(int i, float f, float g) asb ...
14     { ... }
15  ...
```

Figure 7.12: Functional Reduction: Specification of representative reduction method.

#### 7.2.2.2 Reduction over the Representative Invocations

Once the representative invocation of each arm has been determined, a reduction over the representatives determines the actual invocation to service. The method to use during this reduction is specified by a `with-over` clause, as shown on line 10 in Figure 7.12 (for brevity, this figure excludes arm reduction methods). As with the `asb` clause, the specified method is constrained in type to have type signature $(Invocation' \times Invocation) \rightarrow Invocation'$. The arguments correspond, again, to the "previous" and "current" invocations, respectively. As before, $\text{Invocation}'$ may be a subclass of `Invocation`.

[7] The Invocation class has no notion of arguments.

Note that the representative reduction method does not have formal arguments corresponding to the "current" invocation's. This is simply because the formal arguments of the set of operations being serviced need not match (as demonstrated by `entryOne` and `entryTwo` in Figure 7.12).

#### 7.2.2.3 Satisfaction of Acceptability Criteria

The Functional Reduction approach provides access to each pending invocation and its arguments as arguments to a method invocation. Only the single invocation returned as a result of the representative reduction is removed from the set of pending invocations and serviced. The approach provides no means to remove or insert additional invocations.

### 7.2.3 Hybrid

The Hybrid approach combines aspects of the Functional Reduction and Invocation Enumeration approaches in an attempt to balance elegance and performance. Under this approach, representative invocations are selected from each arm of the `inni` statement using a reduction method as discussed for Functional Reduction. An `InvocationEnumeration`, representing an enumeration of the representative invocations, is then passed as an argument to the representative selection method. The methods to use are specified as in the respective base approaches (i.e., `asb` for the arm reduction method and `with-over` for the representative selection method). The typing constraints for these methods match those of the base approaches (except that the representative selection method takes an `InvocationEnumeration` argument instead of an `ArmEnumeration`). Within the representative selection method, invocations are extracted from the enumeration and an invocation's arguments are accessed in a manner similar to Named Enumeration. The invocations are implicitly "named" by the return type of the reduction methods rather than by an `as` clause.

An alternative Hybrid approach provides access to the invocations within each operation through enumerations (one per arm) and then selects the final invocation to service through a reduction over the representative invocations. This approach was not explored because part of the elegance of the functional reduction approach is the manner in which access to an invocation's arguments is provided.

#### 7.2.3.1 Satisfaction of Acceptability Criteria

The Hybrid approach provides access to each pending invocation and its respective arguments through the arm reduction method. Access to the underlying implementation is not provided, so multiple invocations cannot be removed and invocations cannot be illegally inserted.

## 7.3 Analysis of Mechanisms

Though the approaches described in Section 7.2 all satisfy the criteria for acceptable invocation selection mechanisms, they differ in terms of abstraction and performance. The approaches are compared using representative examples from three problem domains. The domains are:

- Selection Independent of Arguments:
  Selection does not require access to the actual arguments of the invocations. This domain's representative, RANDOM, selects a random invocation from the set of pending invocations.
- Single Comparison Required:
  A comparison between two invocations is sufficient to eliminate one from consideration for selection. This domain's representative, PRIORITY, selects an invocation based on a "priority" argument. For simplicity, the "priority" argument is the first argument of each invocation.
- Multiple Comparisons Required:
  Selection requires simultaneous examination and comparison of multiple invocations. This domain's representative, MEDIAN, selects the invocation with the median first argument of all pending invocations[8].

### 7.3.1 Selection Independent of Arguments

#### 7.3.1.1 Invocation Enumeration: RANDOM

Figure 7.13 gives the selection method used in an Invocation Enumeration solution to RANDOM. The `select_random` method calculates the total number of pending invoca-

[8]For simplicity, the example code assumes that the median of the first argument appears in one of the invocations.

```
Invocation select_random(ArmEnumeration arm)
{
  int num = 0;

  while (arm.hasMoreElements()) // tally invocations
  {
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum != null)
      num += invoc_enum.length();
  }

  int rand = RandomInvoc.rand.nextInt(num);

  arm.reset(); // return to beginning of enumeration
  while (arm.hasMoreElements()) // find invocation
  {
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum != null)
    {
      if (rand >= invoc_enum.length)
      {
        rand -= invoc_enum.length;
      }
      else
      {
        while (rand > 0) // find invocation in this “arm”
        {
          invoc_enum.nextElement();
          rand--;
        }
        return (Invoc) (invoc_enum.nextElement());
      }
    }
  }

  return null; // Shouldn't get here
}
```

Figure 7.13: Invocation Enumeration: RANDOM.

tions in the enumeration, generates a random number in the range, and, finally, returns the “random” invocation to service. This solution is straightforward and, since the invocation arguments are not accessed, it is valid under both Invocation Enumeration approaches.

```
1  RandomInvoc arm_random_1(RandomInvoc prev, Invocation cur, int i)
2  {
3    if (prev == null)
4    {
5      return new RandomInvoc(cur, 1);
6    }
7    else
8    {
9      prev.num += 1; // num is a field in user-defined RandomInvoc
10     float rand = RandomInvoc.rand.nextFloat();
11     if (rand <= (1.0/prev.num))
12       prev.setInvocation(cur); // predefined Invocation method
13     return prev;
14   }
15 }
16
17 RandomInvoc in_random(RandomInvoc prev, Invocation cur)
18 {
19   if (prev == null)
20   {
21     if (cur == null)
22       return null;
23     else
24       return (RandomInvoc) cur;
25   }
26   else
27   {
28     RandomInvoc old = (RandomInvoc) cur;
29     prev.num += old.num;
30     float rand = RandomInvoc.rand.nextFloat();
31     if (rand <= (((float)old.num)/prev.num))
32       prev.setInvocation(cur);
33     return prev;
34   }
35 }
```

Figure 7.14: Functional Reduction: RANDOM.

#### 7.3.1.2 Functional Reduction: RANDOM

Figure 7.14 gives a solution to RANDOM using the Functional Reduction approach. In this solution, a representative invocation is selected from each arm through a reduction using the respective `arm_random_x` method. This method randomly determines if the current invocation should be the representative invocation (lines 9–13). As an optimization, line 12 uses the `setInvocation` method of the `Invocation` class to update bookkeeping information within the invocation referred to by `prev`[9]. This information is used to maintain a link between a user-defined invocation and an actual invocation associated with an operation. Once the representative invocations have been selected, the invocation to service is determined during a reduction over the set of representative invocations using `in_random` (lines 17–35).

The Functional Reduction solution to RANDOM requires a method call per invocation to determine the representative invocations and a method call per arm to select the invocation to service. In addition, this solution requires a user-defined invocation class to store information used during the reductions.

```
Invocation select_random(InvocationEnumeration invoc_enum)
{
  int num = 0;

  while (invoc_enum.hasMoreElements()) // tally invocations
  {
    RandomInvoc invoc =(RandomInvoc)invoc_enum.nextElement();
    if (invoc != null)
      num += invoc.num;
  }

  int rand = RandomInvoc.rand.nextInt(num);

  invoc_enum.reset();
  while (invoc_enum.hasMoreElements()) // find invocation
  {
    RandomInvoc invoc =(RandomInvoc)invoc_enum.nextElement();
    if (invoc != null)
    {
      if (rand > invoc.num)
        rand -= invoc.num;
      else
        return invoc;
    }
  }
  return null; // Shouldn't get here
}
```

Figure 7.15: Hybrid: RANDOM.

[9] One could return a newly instantiated object as on line 5, but doing so would be less efficient.

#### 7.3.1.3 Hybrid: RANDOM

A solution to RANDOM using the Hybrid approach is a logical mix of the two previous solutions. The solution selects a representative invocation from each arm using a reduction (as in Functional Reduction) and then selects the final invocation to service from an enumeration of the representative invocations (similar to Invocation Enumeration). In fact, the arm reduction methods are identical to those for the Functional Reduction approach. Figure 7.15 shows the method that selects the final invocation to service.

The Hybrid solution to RANDOM still requires a number of method calls equal to the number of invocations. The reduction over representative invocations, however, is replaced by an enumeration of the representative invocations. The method that selects the final invocation to service need not, in general, examine every invocation in the enumeration.

It must be noted that the Invocation Enumeration solutions to RANDOM implement a more efficient algorithm than the Functional Reduction and Hybrid solutions. The Invocation Enumeration solutions use a divide-and-conquer approach to examine, in the average case, one twentieth of the pending invocations. An operation's entire set of pending invocations is skipped if the random invocation is not a member of the set (see lines 20-32 in Figure 7.13). It is interesting to note that the Functional Reduction and Hybrid approaches cannot implement the divide-and-conquer algorithm employed in the Invocation Enumeration solutions because the arm reduction examines every invocation.

### 7.3.2 Single Comparison Required

#### 7.3.2.1 Invocation Enumeration: PRIORITY

Figure 7.16 gives the selection method used in a View Enumeration solution to PRIORITY. The algorithm used is very simple: loop through the invocations for each arm and record the invocation with the highest priority. A `view` statement is used to gain access to the arguments of each invocation (line 15). `View` statements provide sufficiently abstract access to an invocation's arguments, but the structure of a `view` must closely match that of the associated `inni` statement (e.g., the invocation types used in a `view` directly correspond to the operations serviced by the `inni`). This creates a strong link between an `inni` statement and a selection method, potentially preventing reuse of the method.

The corresponding Named Enumeration solution replaces the `view` statement on lines 15–17 in Figure 7.16 with `instanceof` checks against user-defined types (see Fig-

```
Invocation prio_select(ArmEnumeration arm_enum)
{
  Invocation cur = null;
  int best_prio = 0, cur_prio;

  while (arm_enum.hasMoreElements())
  {
    InvocationEnumeration invoc_enum = arm_enum.nextElement();
    if (invoc_enum == null) continue;

    while (invoc_enum.hasMoreElements())
    {
      Invocation invoc = invoc_enum.nextElement();

      view invoc as (int prio_tmp, int i) //params of invoc
        cur_prio = prio_tmp;
      ... // for each invocation type

      if ((cur == null) || (cur_prio < best_prio))
      {
        cur = invoc;
        best_prio = cur_prio;
      }
    }
  }
  return cur;
}
```

Figure 7.16: View Enumeration: PRIORITY.

```
if (invoc instanceof PrioOne)
  cur_prio = ((PrioOne)invoc).priority;
... // for each invocation type
```

```
class PrioOne extends Invocation
{
  public int priority;
  public PrioOne(int priority, int i)
  { this.priority = priority; }
}
```

Figure 7.17: Named Enumeration: PRIORITY.

ure 7.17). An invocation's arguments are accessed through user-given names (as defined in `PrioOne` in Figure 7.17) rather than through parameters provided by a `view` statement. As such, the link between the selection method and the `inni` statement is not as structured. The reduced structural similarity increases the potential for reuse of the selection method.

#### 7.3.2.2 Functional Reduction: PRIORITY

Figure 7.18 shows the arm reduction method of the Functional Reduction solution to PRIORITY. The representative reduction method is almost identical and is, therefore,

```
1   PrioInvoc arm_select_1(PrioInvoc prev, Invocation cur, int prio, int i)
2   {
3     if (prev == null)
4     {
5       return new PrioInvoc(cur, prio);
6     }
7     else
8     {
7       if (prio < prev.getPrio())
8         prev.setInvocation(cur, prio);
9       return prev;
10    }
11  }
```

Figure 7.18: Functional Reduction: PRIORITY.

not shown. Each method call simply compares the priority of the “current” invocation against that of the current representative invocation (i.e., `prev`) and returns the invocation with higher priority. An invocation’s actual arguments are easily accessible through the parameters passed to the arm reduction method. A representative invocation’s arguments are accessed through fields or methods of the user-defined invocation object (as seen on line 7 in Figure 7.18).

#### 7.3.2.3 Hybrid: PRIORITY

The Hybrid solution to PRIORITY is, again, a logical combination of the two previous solutions. The arm reduction method is identical to that given in Figure 7.18. The representative selection method, shown in Figure 7.19, closely mirrors the selection method for the Invocation Enumeration solution. Under the Hybrid approach, however, the enumeration only provides access to the representative invocations. Within the representative selection method, invocation arguments are accessible through the user-defined invocation objects returned by the arm reduction methods.

```
Invocation prio_select(InvocationEnumeration invoc_enum)
{
  Invoc cur = null;
  int best_prio = 0, cur_prio;

  while (invoc_enum.hasMoreElements())
  {
    PrioInvoc invoc = (PrioInvoc)invoc_enum.nextElement();
    if (invoc == null) continue;

    cur_prio = invoc.getPrio();
    if ((cur == null) || (cur_prio ¡ best_prio))
    {
      cur = invoc;
      best_prio = cur_prio;
    }
  }
  return cur;
}
```

Figure 7.19: Hybrid: PRIORITY.

```
Invocation median_select(ArmEnumeration arm)
{
  Vector v = new Vector();

  while (arm.hasMoreElements())
  {
    InvocationEnumeration invoc_enum = arm.nextElement();
    if (invoc_enum == null) continue;

    view invoc as (int firstarg, int i) // params of invoc
      v.add(new Element(invoc, firstarg));
    ... // for each invocation type
  }
  ... // convert vector, sort, and return median invocation
}
```

Figure 7.20: View Enumeration: MEDIAN.

### 7.3.3 Multiple Comparisons Required

#### 7.3.3.1 Invocation Enumeration: MEDIAN

Figure 7.20 shows the (abbreviated) selection method used in a View Enumeration solution to MEDIAN. The algorithm used gathers the invocations (stored with their first argument in an `Element` object) into a `Vector`, converts the `Vector` into an array, sorts the array, and, finally, selects the median invocation. This approach uses a `view` statement to access the actual arguments of the individual invocations. As in PRIORITY, the Named Enumeration solution differs only in the naming of invocation types and the use of `instanceof` and casting to access invocation arguments.

```
FuncInvoc arm_median_1(FuncInvoc prev, Invocation cur, int firstarg, int i)
{
  if (prev == null)
  {
    return new FuncInvoc(cur, firstarg);
  }
  else
  {
    prev.addInvoc(cur, firstarg);
    return prev;
  }
}
ArmFuncInvoc median_select(ArmFuncInvoc prev,Invocation cur)
{
  if (prev == null)
  {
    if (cur == null)
    {
      return null;
    }
    else
    {
      ArmFuncInvoc ret = new ArmFuncInvoc((FuncInvoc)cur);
      ret.selectInvocation(); //select invocation to service
      return ret;
    }
  }
  else
  {
    prev.addInvoc((FuncInvoc)cur);
    prev.selectInvocation(); // select invocation to service
    return prev;
  }
}
```

Figure 7.21: Functional Reduction: MEDIAN.

#### 7.3.3.2 Functional Reduction: MEDIAN

A Functional Reduction solution to MEDIAN is shown in Figure 7.21. This solution looks simple, but it actually highlights the drawbacks of this approach. The arm reduction method simply gathers each invocation into a `Vector` stored within the user-defined `FuncInvoc` object (line 9). It is necessary to gather the invocations in this manner because the Functional Reduction approach does not provide direct simultaneous access to all of the invocations.

With the invocations for each arm gathered into `FuncInvoc` objects (defined in Figure 7.22 on lines 1–16), a reduction over the arms selects the final invocation to service. Unfortunately, since a specific call of the reduction method cannot determine if it is the last, each call of the reduction method must select the "final" invocation to service (i.e., the median invocation of the set of invocations examined thus far[10]). As such, each call of the representative reduction method performs the costly selection of the median invocation (Figure 7.22, lines 27–29).

#### 7.3.3.3 Hybrid: MEDIAN

The representative selection method for the Hybrid solution to MEDIAN is given in Figure 7.23. The arm reduction method is, again, identical to that of the Functional Reduction solution. Unlike Functional Reduction, however, this solution executes the median selection algorithm only once.

### 7.3.4 Performance

A number of experiments were run to evaluate the performance of each of the proposed mechanisms. The results of these experiments aided in the selection of the mechanism to support in JR. Since the JR compiler did not support any of the proposed mechanisms, these experiments used "hand" translations of the solutions from their respective proposed JR syntax to standard Java. The experiments were conducted on a 850 MHz Intel Pentium III with 128 MB of RAM running Linux kernel 2.2.19 and IBM's Java Runtime Environment 1.3.0.

The experiments presented below apply the solutions presented in the previous discussion to an `inni` statement with 10 arms servicing 10 distinct operations. The initial

[10]This is similar to a loop invariant stating that an array is sorted up to the index of the current iteration.

```
class FuncInvoc extends Invocation
{
  Vector v;
  public FuncInvoc(Invocation invoc, int i)
  {
    super(invoc);
    v = new Vector();
    v.add(new Element(invoc, i));
  }
  public void addInvoc(Invoc invoc, int i)
  { v.add(new Element(invoc, i)); }
  public Enumeration elements()
  { return v.elements(); }
  public Collection getCollection()
  { return v; }
}

class ArmFuncInvoc extends Invoc
{
  Vector v;

  public ArmFuncInvoc(FuncInvoc invoc)
  { v = new Vector(invoc.getCollection()); }
  public void addInvoc(FuncInvoc invoc)
  { v.addAll(invoc.getCollection()); }
  public void selectInvocation()
  {
    ... // convert vector, sort, and return median invocation
  }
}
```

Figure 7.22: Functional Reduction: Support classes.

number of pending invocations was varied from 10 to 100,000 (for RANDOM and PRIORITY) and from 10 to 40,000 (for MEDIAN). Prior to the first execution of the `inni` statement, the pending invocations were evenly distributed over the operations. The process of selecting and servicing an invocation was repeated until all pending invocations were serviced. In all of the experiments, the block of code to service an invocation was left empty. As such, the execution time measured is only that incurred by selecting and removing an invocation from the set of pending invocations.

```
Invocation median_select(InvocationEnumeration invoc_enum)
{
   Vector v = new Vector();
   while (invoc_enum.hasMoreElements())
   {
      FuncInvoc invoc = (FuncInvoc) invoc_enum.nextElement();
      if (invoc == null) continue;
      v.addAll(invoc.getCollection());
   }

   ... // convert vector, sort, and return median invocation
}
```

Figure 7.23: Hybrid: MEDIAN.

#### 7.3.4.1 RANDOM

Figure 7.24 plots, for each solution to RANDOM, the total number of initially pending invocations versus the time to select and service all invocations. This graph demonstrates that, for RANDOM, the Functional Reduction and Hybrid approaches do not compare favorably to the (identical solutions for the) Invocation Enumeration approaches. As discussed in Section 7.3.1, the Invocation Enumeration solutions implement a more efficient algorithm than can be implemented using either the Functional Reduction or the Hybrid approach. Figure 7.25 restricts the plots to the number of invocations that typical programs are likely to service.

For comparison, the graphs also plot the performance of two additional Invocation Enumeration solutions labeled `Naive` and `Fully Iterative`. The solution labeled `Naive` examines every invocation within each operation's set until it finds the random invocation; the `Naive` solution does not skip the sets in which the invocation does not appear. The `Fully Iterative` solution examines every pending invocation, continuing even after finding the random invocation. The Functional Reduction and Hybrid solutions compare more favorably to the Fully Iterative solution as the algorithms used in these solutions are very similar.

This experiment was repeated with the number of arms serviced set equal to 1, 2, and 5. The results of these additional experiments closely match those presented in Figure 7.24 with the exception of the Invocation Enumeration divide-and-conquer solutions.

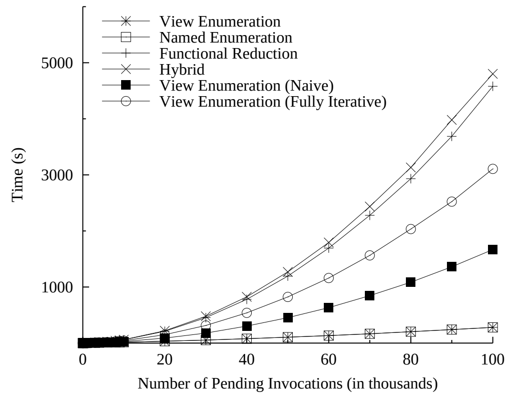



Figure 7.24: RANDOM: Time to service all pending invocations (complete range).

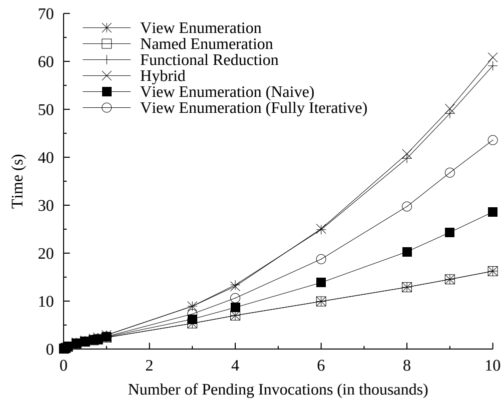



Figure 7.25: RANDOM: Time to service all pending invocations (reduced range).

As the number of arms increases, the Invocation Enumeration solutions perform better because the number of invocations that must be examined decreases[11].

#### 7.3.4.2 PRIORITY

Figure 7.26 plots the total number of initially pending invocations versus the time to select and service all invocations for each solution to PRIORITY. This example is the first in which access to the arguments of the invocations is required and illustrates why the Named Enumeration solution performs so poorly. To support the abstraction provided by the Named Enumeration approach, an object (of the type "named" by the programmer)

[11]In the average case, the number of invocations examined is $1/(2 * N)$, where $N$ is the number of arms being serviced.

must be created for each pending invocation. These extra object creations account for the difference in performance between the two Invocation Enumeration approaches. Though the enumeration phase of Hybrid is similar to Named Enumeration, it does not require, in this example, the creation of a new object for each invocation and, thus, does not suffer the same performance penalty. Figure 7.27 restricts the plots to the number of invocations that typical programs are likely to service.

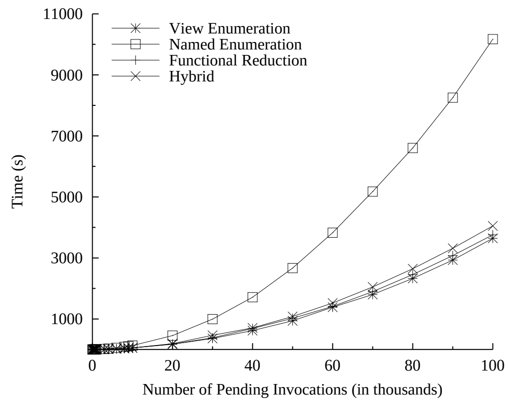



Figure 7.26: PRIORITY: Time to service all pending invocations (complete range).

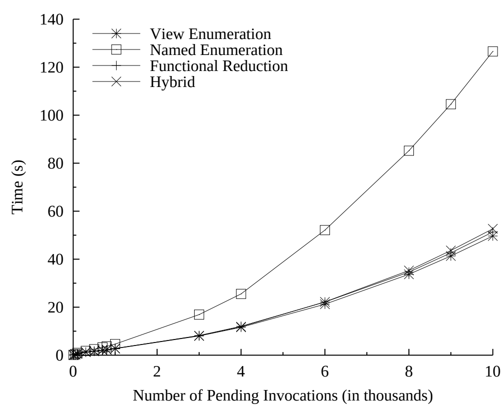



Figure 7.27: PRIORITY: Time to service all pending invocations (reduced range).

This experiment was repeated with the number of arms serviced set equal to 1, 2, and 5. The results of the additional experiments closely match those presented in Figure 7.26.

#### 7.3.4.3 MEDIAN

Figure 7.28 plots the total number of initially pending invocations versus the time to select and service all invocations for each solution to MEDIAN. The Functional Reduction approach performs poorly because the expensive selection computation (sorting the invocations and selecting the median element) is executed for each call of the representative reduction method (equal to the total number of arms). The Hybrid approach performs significantly better because it eliminates this repeated calculation. Figure 7.29 restricts the plots to the number of invocations that typical programs are likely to service.

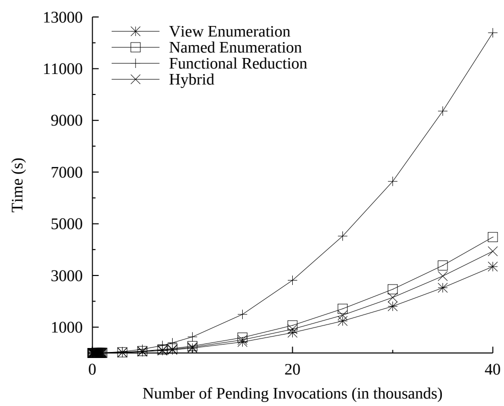



Figure 7.28: MEDIAN: Time to service all pending invocations (complete range).

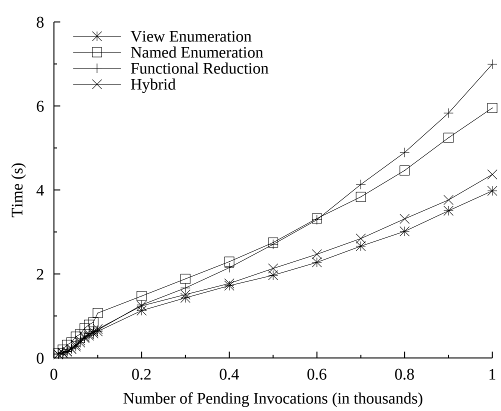



Figure 7.29: MEDIAN: Time to service all pending invocations (reduced range).

This experiment was repeated with the number of arms serviced set equal to 1, 2, and 5. The results of these additional experiments closely match those presented in

Figure 7.24 with the exception of the Functional Reduction approach. As the number of arms is increased, the performance of the Functional Reduction approach greatly worsens due to the repeated execution of the "final" invocation selection during the representative reduction (as discussed in Section 7.3.3.2).

## 7.4 Discussion

The View Enumeration approach performs the best. The `view` statement incurs little overhead compared with direct access to the underlying operation implementation, while providing sufficient abstraction of that implementation. Unfortunately, the structure of a `view` statement must closely match that of the associated `inni` statement (especially when the `inni` statement services operations with the same signature). This structural dependence potentially limits selection method reuse under View Enumeration.

The Named Enumeration approach reduces the structural dependence of the selection method on the `inni` statement by providing a means for the programmer to specify the invocation type as seen by the selection method. This slight decoupling can increase the reuse potential of the selection method. Unfortunately, this approach suffers a large performance penalty because a new object must be created for each invocation. To improve performance, the generic `Invocation` class can provide an access method that returns an invocation's arguments in an `Object` array. Unfortunately, the programmer is then burdened with manually extracting the invocation's arguments from this array. This method improves performance, but reduces abstraction.

The Functional Reduction approach is very elegant, but can suffer greatly in terms of performance. Performance penalties for this strategy can be categorized into method call overhead, repeated selection, and state maintenance. This strategy suffers from the overhead of a method call for each invocation, even when the invocation to service has already been found (e.g., when implementing an algorithm that selects the first invocation with a positive argument). In such a situation, the number of method calls could be minimized if it were possible to abort the reduction. Such a pruning mechanism could inform the system that the current representative invocation is the final invocation for the arm or for the entire `inni` statement. Even with such a mechanism, the worst case number of method calls remains unaffected.

The second performance penalty is repeated selection. Repeated selection, as seen in the MEDIAN solution, is the repeated execution of selection code to satisfy an invariant. In the MEDIAN example, the code to determine the median invocation was repeatedly executed during the representative reduction because it was not possible to determine which method call was the last of the reduction. If such a predicate is provided, then repeated selection can be replaced by repeated tests of the predicate.

A third drawback of Functional Reduction is the need to explicitly carry extra state through a reduction. As demonstrated in the MEDIAN solution, in order to examine more than two invocations at once, the invocations had to be stored as part of the result of each method call. This, unfortunately, makes it necessary to create extra objects, which can adversely affect performance.

Functional Reduction, however, requires less coupling between reduction methods and the `inni` statement being serviced than does View Enumeration. An arm reduction method is only tied to an operation's signature, not to a specific operation. Such methods can can be reused in selection algorithms that require the same reduction over an arm and that expect the same resulting invocation type. Representative reduction methods, however, may require knowledge of the structure of the `inni` statement, as well as knowledge of the arm reduction methods used, in order to differentiate between the types of the representative arm invocations.

The Hybrid approach benefits from the abstractions provided by Functional Reduction, but suffers from the same method call overhead during arm reductions, may require state maintenance during reductions (as for MEDIAN), and may suffer from repeated selection during arm reductions[12]. Of course, the improvements just discussed for Functional Reduction apply equally as well to the reduction phase of the Hybrid approach. Furthermore, with the second reduction phase of the Functional Reduction approach replaced by a single method call, the Hybrid approach has the potential for better performance (as demonstrated for MEDIAN).

The JR programming language supports the View Enumeration approach. Named Enumeration is interesting, but suffers significant performance penalties when implementing selection algorithms in the class of PRIORITY. Similarly, Hybrid performs nearly as well on PRIORITY and MEDIAN, but suffers when it is not necessary to examine every invocation (as in RANDOM). Functional Reduction performs very poorly on MEDIAN because of re-

[12]Such repeated selection can be translated into a gather and a single selection by the representative selection method.

peated selection. In terms of abstraction and performance, the View Enumeration approach is the most promising.

## 7.5 Summary

This chapter discussed different candidate selection mechanisms considered for addition to the JR programming language. For consideration, a candidate had to satisfy criteria requiring selection of any invocation from the set of pending invocations, access to each invocation's actual arguments, prevention of the removal of multiple invocations, and prevention of the insertion of invocations into the pending set. The candidates were evaluated in terms of programming abstraction and performance on representative examples from different problem domains.

Of the candidates, we deemed the View Enumeration approach the most appropriate (in terms of programmer-level abstraction and performance) and extended the JR programming language with support for it. Because the View Enumeration approach does not rely on the subclassing of the `Invocation` class, this approach can also be used in concurrent languages that are not object-oriented. Performance results for each of the experiments using JR with the View Enumeration mechanism are consistent with the results for the "hand" translations discussed in Section 7.3.4.

# Chapter 8

# Conclusion

This chapter reflects on the work presented and concludes the dissertation. It first summarizes the dissertation work and it then discusses the greater applicability of the presented research. Finally, the chapter concludes with a discussion of future work.

## 8.1 Summary

This dissertation has presented three topics of research: the JR programming language, exception handling during asynchronous invocation, and the design and analysis of an inter-operation invocation selection mechanism. The design and implementation of the initial version of the JR programming language formed a foundation on which the remaining research was built and from which the current JR programming language evolved.

### 8.1.1 The JR Programming Language

The JR programming language is a concurrent object-oriented programming that is a result of integrating the SR concurrency model with the Java programming language. JR extends the syntax of Java to provide support for remote virtual machine creation, remote object creation, and remote method invocation, dynamic thread creation, asynchronous method invocation, and multi-way receive. The resulting language supports a rich concurrency model for writing tightly-coupled concurrent programs and for writing reusable concurrent libraries. Moreover, the use of operations and operation capabilities provides support for programs with dynamic communication paths.

The JR programming language adopts, from SR, the operation as its key concurrency abstraction. Operations are defined, like methods, within classes. Operations can be invoked either synchronously or asynchronously and can be serviced by either a method or a multi-way receive (an input statement). JR defines a novel inheritance policy for operations that mirrors the inheritance of methods, but differs because of the two types of operations.

### 8.1.2 Exception Handling during Asynchronous Invocation

Exception handling models for sequential programming languages rely upon the call stack for exception propagation. This propagation method is inadequate for languages with support for asynchronous method invocation. An exception thrown from a method invoked asynchronously cannot, in general, propagate up the call stack because the invoking method may no longer be executing.

The work presented in this dissertation provides support for exception handling during asynchronous invocation through the specification of *handler* objects. Any asynchronous invocation of a method that may throw exceptions must specify an appropriate *handler* object (with appropriate *handler* methods) to which any thrown exceptions will be propagated. This research extends previous work on exception handling during asynchronous invocation with support for exception handling after early replies and after forwarding. Furthermore, within the JR language, such exceptions can be handled by operations in addition to methods.

### 8.1.3 Inter-operation Invocation Selection Mechanism

Many languages support message-passing as a form of communication between threads. These languages, however, differ in the support provided for selecting specific messages during a multi-way receive. Such support ranges from non-deterministic selection, to simple single-message based filtering, to operation (i.e., entry) preferences.

The research presented in this dissertation proposes and analyzes different inter-operation invocation selection mechanisms. Each of these mechanisms allows message selection algorithms to examine every pending message when determining which message to receive. It is not possible to express such selection algorithms using the pre-existing mechanisms supported by other languages. As mentioned above, these other languages allow, at best, single-message based selection[1], in the general case.

The proposed mechanisms are compared in terms of programmer-level abstraction (especially when accessing the arguments on a specific message) and performance on a representative set of selection algorithms. The result of the research was the determination of the most appropriate selection mechanism to support (View Enumeration) and the implementation of this mechanism as part of the JR programming language.

## 8.2 General Applicability

This dissertation presents, separately, three facets that, as a whole, make up the JR programming language. Each of these facets is applicable, in some form, to other programming languages and systems.

### 8.2.1 The JR Programming Language

Chapter 5 presents the design and implementation of the JR programming language. In particular, the chapter discusses the integration of the SR concurrency model with the Java programming language (the host language). Though the implementation benefits from and leverages Java's support for platform-independence, remote method invocation, threading, and reflection, the design of the JR programming language can be used as a basis for integrating SR's concurrency model with any class-based object-oriented programming language (e.g., C++ [78] or C# [58]).

The implementation of a language integrating SR's concurrency model with another host language may require the implementation of a run-time system with support beyond that of the JR run-time system. In particular, the JR implementation benefits from Java's virtual machines, remote method invocation (including marshaling and unmarshaling), threading, and reflection. The implementation of an SR-C++ language, for instance, would require support for threading and remote communication. Such support can be provided through either direct implementation or additional libraries.

[1] A binary selection in which an invocation is either selected or not; invocations can not be compared.

### 8.2.2 Exception Handling during Asynchronous Invocation

Chapter 6 presents an exception handling model that accounts for exceptions thrown during asynchronous invocation. In general, this exception model provides mechanisms to handle exceptions propagated to a split in an invocation chain. The presentation in Chapter 6 discussed not only handling exceptions thrown during asynchronous invocation, but also handling exceptions thrown after early replies or invocation forwarding.

The presented exception handling model is applicable to programming languages that support exceptions and asynchronous invocation, early reply, or forwarding. Moreover, the presented model is applicable to distributed programming systems (e.g., CORBA [62]) that support exceptions and asynchronous invocation. The presented model, however, requires handler *objects* to handle the thrown exceptions. In languages or systems that do not support objects in any form, a list of handler functions could be substituted.

### 8.2.3 Inter-operation Invocation Selection Mechanism

The presented research examines various approaches to inter-operation invocation selection. The most appropriate mechanism for JR was selected and implemented. Other programming languages supporting multi-way receives can readily adapt the selected mechanism, View Enumeration, or one of the other proposed mechanisms. In terms of general applicability, the View Enumeration approach is promising because it does not depend on the object-oriented features of JR. As such, this approach can be supported by concurrent languages that are not object-oriented (e.g., SR).

## 8.3 Future Work

This dissertation work has focused on the design and implementation of the JR programming language and the extension of the JR language to fully integrate exception handling and asynchronous invocation and to support increased flexibility in invocation selection. Experimental evaluation of the JR programming language has been limited to micro-benchmarks (see Sections 5.4 and 7.3.4). These early results are promising. Future work will further evaluate the performance and utility of JR through the design and implementation of realistic distributed systems. Such evaluations will likely highlight opportunities for high-level optimizations.

Further work will study the utility of the proposed exception model. In particular, the use of operations and input statements to service exceptions thrown during asynchronous invocation provides the potential for increased flexibility and may provide an interesting comparison to languages that provide support for concerted exception handling [46].

Finally, the supported inter-entry invocation selection mechanism has only been evaluated using representative examples and compared to the other proposed mechanisms. Studies into the use of the inter-entry invocation selection mechanism may yield servicing patterns and optimizations for common uses.

# Bibliography


[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Superweb: Towards a global web-based parallel computing infrastructure. In *11th International Parallel Processing Symposium*, pages 100–106, 1997.

[2] G. R. Andrews and R. A. Olsson. The evolution of the SR language. *Distributed Computing*, 1(3):133–149, 1986.

[3] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* The Benjamin/Cummings Publishing Co., Redwood City, California, 1993.

[4] 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.

[5] G. R. Andrews and F. B. Schneider. Concepts and notation for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

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

[7] M. S. Atkins. Experiments in SR with different upcall program structures. *ACM Transactions on Computer Systems*, 6(9):365–392, November 1988.

[8] M. S. Atkins and R. A. Olsson. Performance of multi-tasking and synchronization mechanisms in the programming language SR. *Software – Practice and Experience*, 18(9):879–895, September 1988.

[9] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel

programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[11] C. Baquero, R. Oliveira, and F. Moura. Integration of concurrency control in a language with subtyping and subclassing. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, pages 173–184, 1995.

[12] 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.

[13] A. J. Bernstein. Predicate transfer and timeout in message passing systems. *Information Processing Letters*, 24(1):43–52, January 1987.

[14] Jean Bezivin. Some experiments in object-oriented simulation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 394–405, 1987.

[15] A. P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193, 1985. In *ACM Operating Systems Review* 19(5).

[16] T. Brecht, H. Sandhu, J. Talbot, and M. Shan. Paraweb: Towards world-wide supercomputing. In *European Symposium on Operating System Principles*, 1996.

[17] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[18] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[19] Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129, Nottingham, July 10-14 1989. Cambridge University Press.

[20] Jean-Pierre Briot. Actalk: A framework for object-oriented concurrent programming - design and experience. In *Object-Based Parallel and Distributed Computing II – Proceedings of the 2nd France-Japan Workshop*, 1999.

[21] A. Burns. *Programming in occam.* Addison Wesley, 1988.

[22] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[23] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997.

[24] M. Chung and R. A. Olsson. New mechanisms for invocation handling in concurrent programming languages. *Computer Languages*, 24:254–270, December 1998.

[25] P. R. Cook. *mod–a language for distributed programming. *IEEE Transactions on Software Engineering*, 6(6):563–571, November 1980.

[26] Iain Craig. *The Interpretation of Object-oriented Programming Languages.* Springer Verlag, London, England, 1999.

[27] S. Dorward and R. Pike. Programming in Limbo. In *Proceedings of the IEEE Compcon 97 Conference*, pages 245–250, 1997.

[28] Edinburgh Parallel Computing Centre. Java Grande Forum Benchmark Suite. `http://www.epcc.ed.ac.uk/research/javagrande/benchmarking.html`.

[29] T. Elrad and F. Maymir-Ducharme. Distributed languages design: constructs for controlling preferences. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 176–183, St. Charles, Illinois, August 1986.

[30] T. Elrad and F. Maymir-Ducharme. Satisfying emergency communication requirements with dynamic preference control. In *Proceedings of Sixth Annual National Conference on Ada Technology*, March 14-17 1988.

[31] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[32] Benoît Garbinato, Pascal Felber, and Rachid Guerraoui. Protocol classes for designing reliable distributed environments. In Pierre Cointe, editor, *ECOOP '96—Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 316–343. Springer, 1996.

[33] Benoît Garbinato and Rachid Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 221–232, Berkeley, CA, USA, June 1997. USENIX.

[34] N. Gehani and W.D. Roome. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Transactions on Software Engineering*, 14(11):1546–1553, November 1988.

[35] N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.

[36] N. Gehani and W.D. Roome. Message passing in Concurrent C: Synchronous versus asynchronous. *Software – Practice and Experience*, 20(6):571–592, June 1990.

[37] J. B. Goodenough. Exception handling issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[38] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996. ISBN 0-201-63451-1.

[39] Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, October/December 1997.

[40] S. J. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

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

[42] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[43] D. T. Huang. An exception handling mechanism for SR. *Computer Languages*, 15(3):163–176, 1990.

[44] Y. Ichisugi and A. Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Proceedings of the UK/Japan Workshop on Concurrency: Theory, Language, and Architecture*, pages 604–615, 1990.

[45] Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/Ada- IC/standards/95lrm_rat`.

[46] V. Issarny. An exception handling model for parallel programming and its verification. In *Proceedings of the ACM SIGSOFT 1991 Conference on Software for Critical Systems*, pages 92–100, 1991.

[47] V. Issarny. An exception handling mechanism for parallel object-oriented programming: toward reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–40, 1993.

[48] 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.

[49] D. G. Kafura and K. H. Lee. Inheritance in Actor-based concurrent object-oriented languages. In *Proceedings of ECOOP '89*, pages 131–145, 1989.

[50] L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel Java with global object space. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 235–244, 1997.

[51] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 575–584, April 2001.

[52] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of remote procedure call and static process structure for distributed computing. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1986.

[53] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, 1999.

[54] S. Marlow, S. Peyton Jones, and A. Moran. Asynchronous exceptions in Haskell. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285. ACM SIGPLAN, June 2001.

[55] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, 1993.

[56] C. M. McNamee and W. A. Crow. Inter-entry selection control mechanisms: Implementation and evaluation. *Computer Languages*, 22(4):259–278, 1996.

[57] C. N. Mendis. Java-synchronising resources for concurrent and distributed programming. Master's thesis, University of Warwick, 1997.

[58] Microsoft. *C# Language Specification.* Microsoft, Redmond, WA, 2001. http://msdn.microsoft.com/.

[59] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 85–103, 1997.

[60] N. Nagaratnam, A. Srinivasan, and D. Lea. Remote objects in Java. In *IASTED International Conference on Networks*, 1996.

[61] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 152–159, 1999.

[62] The Object Management Group. *The Common Object Request Broker*, 1995. `http://www.omg.org/`.

[63] R. A. Olsson. Using SR for discrete event simulation: A study in concurrent programming. *SOFTWARE — Practice and Experience*, 20(12):1187–1208, December 1990.

[64] R. A. Olsson, G. D. Benson, T. Ge, and A. W. Keen. Fairness in shared invocation servicing. Submitted for publication.

[65] R. A. Olsson and C. M. McNamee. Inter-entry selection: Non-determinism and explicit control mechanisms. *Computer Languages*, 17(4):269–282, 1992.

[66] R. Pandey and J. C. Browne. CYES-C++: A concurrent extension of C++ by compositional mechanisms. In *Workshop on Object-based Concurrent Systems*, October 1994.

[67] R. Pandey and J. C. Browne. Support for implementation of evolutionary concurrent systems. *International Journal of Parallel Programming*, 29(4):401–431, August 2001.

[68] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

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

[70] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

[71] 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.

[72] M. L. Scott. Messages vs. remote procedures is a false dichotomy. *ACM SIGPLAN Notices*, 18(5):57–62, May 1983.

[73] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.

[74] M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.

[75] A. C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[76] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[77] R. E. Strom et al. *Hermes: A Language for Distributed Computing.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[78] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. ISBN 0-201-88954-4.

[79] Sun Microsystems. *Java Remote Method Invocation Specification*. Sun Microsystems, Palo Alto, CA, 1997.

[80] Sun Microsystems. *Java Message Service Specification*. Sun Microsystems, Palo Alto, CA, 2001.

[81] A. Szalas and D. Szczepanska. Exception handling in parallel computations. *ACM SIGPLAN Notices*, 20(10):95–104, October 1985.

[82] J. D. Ullman. *Elements of ML Programming*. ML97 Edition. Prentice Hall, 1998. ISBN 0-13-790387-1.

[83] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportion-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, Monterey, California, November 1994. USENIX.

[84] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger*, 1(1):7–87, 1990.

[85] S. Yemini and D. M. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, 1985.

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