An Exception Handling Mechanism for the Concurrent Invocation Statement

Hiu Ning (Angela) Chan[1], Esteban Pauli[1], Billy Yan-Kit Man[1], Aaron W. Keen[2], and Ronald A. Olsson[1]

[1]Department of Computer Science, University of California, Davis, Davis, CA 95616 USA
{chanhn,pauli,many,olsson}@cs.ucdavis.edu

[2]Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407 USA
akeen@csc.calpoly.edu

Abstract. Several concurrent programming languages and systems — e.g., MPI, .NET, and SR — provide mechanisms to facilitate communication between one process and a group of others. One such mechanism is SR's concurrent invocation statement (co statement). It specifies a group of operation invocations and normally terminates when all of its invocations have completed. To make the co statement more flexible, it can specify code in the invoker to execute as each invocation completes or to terminate the entire co statement before all of its invocations have completed. We have added an SR-like co statement to JR. Unlike SR, JR provides exception handling mechanisms, which are integrated with Java's exception handling mechanism. However, JR needs additional mechanisms to deal with sources of asynchrony. The co statement introduces additional such sources of asynchrony for the invocations it initiates. This paper describes the design and implementation of an exception handling mechanism for JR's co statement.

1 Introduction

Communication between one process and a group of others is important in many concurrent programs. Several concurrent programming languages and systems provide mechanisms to facilitate such communication, e.g., MPI's [9] collective communication; .NET's [10] delegates and asynchronous invocations, which can be used to simulate collective communication; and SR's [2, 1] concurrent invocation statement (*co statement*, for short), which can be viewed as a form of collective communication. These mechanisms can be used to broadcast data to a group of processes and gather back results from each in the group, for instance, in applications such as initiating subparts of numerical computations, reading/writing a distributed database, and distributed voting schemes.

This paper focuses on the co statement. It specifies a group of operation invocations and normally terminates when all invocations have completed. The co statement also allows quantifiers to deal with groups of related operations and post-processing code (PPC) that is executed as each invocation completes. The co statement is allowed to terminate before all its invocations have completed. All these features are useful in practice, as seen in the examples in SR [2, 1] and later in this paper.

We have added a co statement, similar to SR's, to the JR concurrent programming language [7, 11]. Unlike SR, JR provides exception handling mechanisms, which are integrated with Java's exception handling mechanism. However, JR needs additional mechanisms to deal with sources of asynchrony, as described in Reference [8]. The concurrent invocation statement introduces additional such sources of asynchrony for the invocations it initiates. Thus, we have also designed and implemented an exception handling mechanism for the concurrent invocation statement and added it to JR (available on the web [5]). Our work should benefit others considering adding exception handing mechanisms to other concurrent programming languages and systems.

The rest of this paper is organized as follows. Section 2 presents the concurrent invocation statement without exceptions, introduces our running example, and summarizes our previous work with handling exceptions during asynchronous method invocation. Section 3 gives an overview of our approach, illustrates it by extending our running example to use exceptions, and discusses and justifies our design decisions. Section 4 presents an overview of our implementation and discusses its reasonable performance. Finally, Section 5 concludes.

2 Background

2.1 Concurrent Invocation Statement (without Exceptions)

Figure 1 shows a small JR program with a simple co statement. It simulates

```
public class main {
  public static void main(String [] args){
    co aliceVote();
    [] bobVote();
    System.out.println("Election is over.");
  }
  public static op void aliceVote() {
    ...
    System.out.println("Alice votes yes.");
    ...
  }
  public static op void bobVote() {
    ...
    System.out.println("Bob votes no.");
    ...
  }
}
```

Fig. 1. A simple election program.

a two-person election, where each person announces his or her vote. This co statement contains two arms. The process executing the co statement initiates the two *co invocations* (one of `aliceVote` and one of `bobVote`); it then waits until both invocations have finished. Thus, the individual "votes" outputs are guaranteed to occur before the "over" output.

Figure 2 shows a more interesting voting program fragment. Here, votes are

```
int yesCount = 0, noCount = 0;  boolean vote[] = new boolean [voters];

co ((int i = 0; i < voters; i++)) vote[i] = getVote[i]();

for (int i = 0; i < voters; i++) {
  if(vote[i]) ++yesCount;  else ++noCount;
}
// announce decision
System.out.println("votes  For: " + yesCount + "  Against:  " + noCount);
if (yesCount > noCount)
  System.out.println("Victory!");
else
  System.out.println("Defeat ;-(");
```

Fig. 2. Election with tallying of votes after voting.

received from an array of voters' operations, `getVote`. This co statement uses quantifier notation to initiate all invocations. As each invocation completes, the result is recorded in the `vote` array. After all votes are received, the votes are tallied and the decision is announced.

Figure 3 shows how votes can be tallied *as they are received*. This co statement

```
boolean vote;

co ((int i = 0; i < voters; i++)) vote = getVote[i](){
  System.out.println("Voter "+i+ " voted "+vote);
  if (vote)
    ++yesCount;
  else
    ++noCount;
}
// announce decision -- same code as in earlier figure
```

Fig. 3. Election with tallying of votes during voting.

specifies *post-processing code (PPC)*; the scope of the quantifier variable for the arm includes the arm's PPC. As each invocation completes, the corresponding PPC is executed by the *same* process that initiates the co statement. Thus, execution of PPCs is serial and variables local to this process used within the PPC (e.g., `vote`) are not subject to race conditions. (The assignment to `vote` is considered a part of the PPC; it is executed before the rest of the PPC.)

The co statement has an iterative nature with respect to it executing its PPCs, so a break statement makes sense within a PPC. Figure 4 shows a co statement that announces the election decision as soon as a majority of voters has decided the election. A co statement now terminates when: all its invocations

```
int yesCount = 0, noCount  = 0;  boolean vote = false;

co ((int i = 0; i < voters; i++)) vote = getVote[i](){
  if (vote) {
    if (++yesCount > voters/2) break;
  }
  else
    if (++noCount >= (voters+1)/2) break; // tie -> No
}
// announce decision -- same code as in earlier figure
```

Fig. 4. Election with decision announced as soon as majority has decided.

have completed and their corresponding PPCs have terminated; or execution of a PPC has executed a transfer of control out of the co statement. Note that, in this example, invocations whose results are not tallied do continue to execute, even if the co statement has terminated. However, their subsequent completion has no effect on the invoking process; indeed, the invoking process may have completed and no longer exist.

2.2 Simulation of the co Statement Using Existing JR Mechanisms

The co statement can be simulated with other JR language mechanisms, but doing so is cumbersome for the programmer. For example, Figure 5 shows how

```
public class main {
  public static void main(String [] args){
    op void aliceVoted();
    op void bobVoted();
    send aliceVote(aliceVoted);
    send bobVote(bobVoted);
    receive aliceVoted();
    receive bobVoted();
    System.out.println("Election is over.");
  }
  public static op void aliceVote(cap void() voted) {
    ...
    System.out.println("Alice votes yes.");
    ...
    send voted();
  }
  public static op void bobVote(cap void() voted) {
    ...
    System.out.println("Bob votes no.");
    ...
    send voted();
  }
}
```

Fig. 5. Hand-coded simulation of Fig. 1.

to rewrite Figure 1. It uses send invocations (which are non-blocking) to initiate the voting and the receive statement to wait until both voters have voted. Here, and in general, the code requires changing the interface to the vote operations to have an extra parameter, so that the voter can notify “election central” when it has finished voting. This extra parameter is a capability (a special kind of reference) to an operation.

The general simulation of a co statement with a PPC is more complicated. To illustrate, Figure 6 shows how to rewrite Figure 4. Figure 6’s code properly simulates Figure 4’s co statement because the PPC code is executed sequentially, but it again requires the interface to be changed, e.g., the signatures of `getVote` and the body of that operation (not shown). Although this simulation works fine for the program in Figure 4, the general simulation is more complicated. For example, suppose the co statement uses the quantifier variable in its PPC, as in Figure 3. At first look, the code in Figure 6 might seem to work, but the index variable in the second for loop (where the print statement would be placed) has no connection to the index variable in the first for loop. Also, if the co statement has multiple arms, a simple receive statement no longer suffices. Section 4.1 discusses how to deal with these problems in general.

```
op void voted(boolean);
for (int i = 0; i < voters; i++) {
  send getVote[i](voted);
}
for (int i = 0; i < voters; i++) {
  boolean vote;
  receive voted(vote);
  if (vote) {
    if (++yesCount > voters/2) break;
  }
  else
    if (++noCount >= (voters+1)/2) break; // tie -> No
}
// announce decision -- same code as in earlier figure
```

Fig. 6. Hand-coded simulation of Fig. 4.

2.3 Handling Exceptions During Asynchronous Method Invocation

This section summarizes our earlier work that shows how to handle exceptions during asynchronous method invocation [8, 11]. Our approach bears some resemblance to that provided in both ABCL/1 [3] and Arche [4].

JR provides asynchronous method invocation via the send statement. 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 out of the invoked method are directed to the handler object. To be used as a handler, an object must implement JR's `Handler` interface and define a number of handler methods. A method is defined as a handler through the use of the `handler` modifier (much like the `public` modifier). A handler method takes 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 the handler method of the appropriate type.

Figure 7 shows an example definition of a handler object's class and how it

```
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 */ }
}

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

Fig. 7. Class definition for and use of a simple handler object.

is used. 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. As seen in Figure 7, a send statement must specify, using a handler clause, its handler object. The JR compiler statically checks that the specified handler object can handle each of the potentially thrown exceptions.

3 Design

3.1 Overview of Exceptions in the Concurrent Invocation Statement

A key observation in integrating exception handling with the concurrent invocation statement is that the co statement adds another source of asynchrony in the same sense as for the send statement (Section 2.3). Section 2.1 described how if the co statement's PPC contains a break statement, then the invoking process may not even exist when one of its invocations completes; the same now also pertains to an exception that occurs for one of its co invocations. Therefore, we add a handler to co invocations that can throw exceptions.

Section 2.1 described when a co statement terminates. A co invocation that throws an exception does *not* cause its associated PPC to be executed (discussed further in Section 3.2). But, that invocation is now considered to have completed and contributes toward the co statement's overall termination.

Figure 8 shows how to extend the program in Figure 4 for when the `getVote`

```
  boolean decided = false;

  MyHandler mh = new MyHandler();

  co ((int i = 0; i < voters; i++)) vote = getVote[i]() handler mh : {
    if (vote){
      if (++yesCount > voters/2) {decided = true; break;}
    }
    else
      if (++noCount >= (voters+1)/2) {decided = true; break;} // tie -> No
  }

  System.out.println("votes  For: " + yesCount + "  Against:  " + noCount);
  if (!decided)
    System.out.println("Too many non-participating voters to decide election");
  else  {
    if (yesCount > noCount)
      System.out.println("Victory!");
    else
      System.out.println("Defeat ;-(");
  }

public class MyHandler implements edu.ucdavis.jr.Handler {
  public handler void handleNonParticipVoter(NonParticipVoterException e) {
    System.out.println("Non-participating voter");
  }
}

public class NonParticipVoterException extends java.lang.Exception {
}
```

Fig. 8. Fig. 4 extended to handle exceptions.

operation can throw exceptions. The co invocation now specifies a handler, which just outputs an error message. The code that outputs the results now makes sure that enough voters actually voted yes or no.

3.2 Design Decisions

A simpler approach than using handler objects (Section 3.1) is to just enclose the co statement, such as the one in Figure 4, within a try/catch statement.

However, if an exception occurs for an invocation, then control transfers to the catch block and the entire co statement terminates. Thus, the results of those invocations that complete normally after the exception occurs would be lost. That would make dealing with code that can throw exceptions, such as that in Figure 8, much more difficult.

Having invocation-specific handlers allows the co statement to continue in such cases and terminate cleanly. Moreover, because the PPC can contain statements such as break, exceptions on invocations must be handled somewhere, as noted in Section 3.1. Thus, if an op can throw an exception, its invocation within an arm of a co statement must have a handler.

Figure 8 illustrated the use of a handler that is specified for each invocation. The co statement also allows a default handler for the entire statement so that exceptions from all invocations are handled by the same handler object. The default handler is used for any invocation that requires a handler but does not itself specify a handler. Allowing a default handler is convenient so that users do not need to specify a handler object for each arm while they can occasionally provide a special handler for some arms to handle their exceptions. The default handler is used only if an invocation-specific handler is not specified for a particular invocation. An alternative is to allow the default handler to be used in addition to the invocation-specific handler, so that it can handle some types of exception that a handler object for a specific invocation cannot handle. However, we have not yet seen a real need for that functionality.

If an exception occurs during execution of a PPC, then execution of the current block will terminate and control transfers out of the co statement, thus terminating the co statement. Consider, for example, the following co statement

```
co f() {
  ... // PPCf -- throws exception (but contains no try/catch)
}
[] g() {
  ... // PPCg
}
```

If the invocation of `f` finishes before the invocation of `g` and `PPCf` throws an exception that is not caught within `PPCf`, then `PPCg` will not be executed when the invocation of `g` finishes. This behavior is consistent with exceptions in Java; e.g., if an exception occurs within a loop in Java code and is not caught within the loop, the rest of the loop is not executed.

If an exception occurs for a co invocation, the associated PPC is not executed. This behavior was seen in Figure 8. In addition, if the co invocation assigns to a variable (`vote` in Figure 8), that assignment is considered part of the PPC and is not executed if an exception occurs. An alternative would, of course, be to allow the PPC to execute, but it would need some way to distinguish between success and exception (e.g., `if "exceptionOccurred"` ...), so that, for example, it would know whether the variable was assigned.

4 Implementation

4.1 Internal Transformations

Section 2.2 showed how simple co statements can be simulated using other JR language mechanisms. Internally, the JR translator transforms a co statement in a way similar to those examples; however, the transformation handles the necessary change of interface and deals with multiple arms and quantifier variables.

For example, the co statement in Figure 9 is translated internally to roughly

```
co f(5) {PPCf} [] g() {break;} [] ((int i = 0; i < N; i++)) x[i] = h(i) {PPCh}
```

Fig. 9. Example co statement.

the code in Figure 10. This transformed code first initiates the invocations of all

```
cap void (void) f_retOp = f.cocall(...);
cap void (void) g_retOp = g.cocall(...);
cap void (void) h_retOp [] = new cap void (void) [N];
for (int i = 0; i < N; i++) { h_retOp[i] = h.cocall(...); }
for (int JR_CO_COUNTER = 0; JR_CO_COUNTER < 2+N; JR_CO_COUNTER++) {
  inni void f_retOp() {PPCf}
  [] void g_retOp() {break;}
  [] ((int i = 0; i < N; i++)) void h_retOp[i](int retVal) {x[i] = retVal; PPCh }
}
```

Fig. 10. Transformed version of Figure 9.

`2+N` operations.[1] Internally, JR operations are objects with methods for various ways of invoking them [7]. The new `cocall` method initiates an invocation of its operation, but it does not block. It returns a capability for an operation that will be invoked when the initiating invocation completes. The code in Figure 10 then uses a loop to wait for all of the co's invocations to complete (i.e., the `_retOp` operations to be invoked), but the loop can be exited early. Its body contains an inni statement, which is JR's multi-way receive statement: each execution of `inni` services an invocation for one of its arms. A multi-way receive is needed here because the order in which the invocations complete is unknown; indeed, not all invocations need to complete for a co statement to terminate, as illustrated in Figure 4. Note how the PPCs in the original program in Figure 9 simply become blocks of code in the `inni` in Figure 10; in particular, the break statement in the co's PPC simply becomes a break statement that now applies to the for loop. Also note how the third arm of the `inni` has a quantifier that is identical to the quantifier in the original program; thus, the quantifier variable's value is the same in the original invocation and in the PPC executed for the corresponding `_retOp` invocation.

The implementation supports exception handlers in co statements by extending the above scheme. For example, consider adding a handler to the invocation of `f` in Figure 9, as shown in Figure 11. If an exception occurs for the invocation

```
MyHandler mh = new MyHandler();
co f(5) handler mh : {PPCf} [] g() {break;} [] ((int i = 0; i < N; i++)) x[i] = h(i) {PPCh}
```

Fig. 11. Example co statement with exception handler.

of `f`, then, without any change in the above scheme, the inni statement in Fig-

[1] The code records for its later use the actual number of co invocations in case any of the expressions in the quantifiers or co invocations have side effects.

ure 10 would block forever waiting for `f_retOp` to be invoked. The extension, then, prevents that by defining an additional operation that is invoked when an exception occurs. Thus, the co statement in Figure 11 is translated internally to roughly the code in Figure 12. Note how operation `co_fail_retOp` is created,

```
MyHandler mh = new MyHandler();
cap void (void) co_fail_retOp = new op void(void); // new op
cap void (void) f_retOp = f.cocall(mh, co_fail_retOp, ...); // extra parameters
cap void (void) g_retOp = g.cocall(...);
cap void (void) h_retOp [] = new cap void (void) [N];
for (int i = 0; i < N; i++) { h_retOp[i] = h.cocall(...); }
for (int JR_CO_COUNTER = 0; JR_CO_COUNTER < 2+N; JR_CO_COUNTER++) {
  inni void f_retOp() {PPCf}
  [] void g_retOp() {break;}
  [] ((int i = 0; i < N; i++)) void h_retOp[i](int retVal) {x[i] = retVal; PPCh }
  [] void co_fail_retOp() {} // new arm -- no body needed
}
```

Fig. 12. Transformed version of Figure 11.

passed with the handler `mh` as extra parameters to the cocall method (which is now overloaded to allow such), and appears in a new arm in the `inni`. If execution of the invocation of `f` throws an exception, then both the handler `mh` is executed and the `co_fail_retOp` operation is invoked; otherwise, only the `f_retOp` operation is invoked.

4.2 Performance

We ran several micro- and macrobenchmarks to assess the performance of our co statement implementation. We ran these benchmarks on various PCs (1.4GHz and 2.0GHz uniprocessors; 2.4GHz and 2.8GHz dual-processors) running Linux; specific results, of course, varied according to platform, but the overall trends were the same.

The benchmarks confirmed that our implementation of the co statement had no noticeable impact on regular invocations; and that the cost of executing a co statement with no exceptions is nearly the same as executing a co statement with exceptions but with no exceptions actually thrown during its execution.

The benchmarks also showed that the execution costs for our implementation of the co statement were nearly identical to the hand-coded simulations of the co statement. For example, the execution costs of the programs in Figures 4 and 6 were about the same for various numbers of voters (10, 100, 500, 1000, 1200, and 1500). (Actual code and execution times are available on the web [6].)

Our initial implementation did not perform as well as our current implementation for larger numbers of voters. It used the more straightforward approach of creating a `_fail_retOp` operation for each operation in the co statement that might throw an exception, including an array of such failure operations for each quantified operation. The cost of allocation of these additional operations was high (e.g., for large numbers of voters or when done repeatedly within a loop), so our current implementation eliminates them.

We considered measuring the performance of our co implementation against those for other language implementations, but as noted in Section 1, only SR provides a co statement and it does not provide exception handling. We could measure the costs of SR's co statement versus JR's co statement without exception handling, but that would really measure more the differences of C (in which SR is implemented) versus Java (in which JR is implemented).

5 Conclusion

We have extended the JR programming language with a concurrent invocation statement that includes support for exception handling. This paper described the design tradeoffs, the implementation and performance of the new exception handling mechanism, and some examples illustrating how to use this mechanism. This new feature has been incorporated into the standard JR language release, which is available on the web [5]. We hope to extend our work by considering more formal semantics for the concurrent invocation statement and exception handling for it, and with more comprehensive performance evaluation.

Acknowledgements

Others in the JR group — Erik Staab, Ingwar Wirjawan, Steven Chau, Andre Nash, Yi Lin (William) Au Yeung, Zhi-Wen Ouyang, Alex Wen, and Edson Wong — assisted with this work. The referees provided thoughtful comments.

References

1. G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.
2. 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.
3. 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.
4. 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.
5. JR distribution. `http://www.cs.ucdavis.edu/~olsson/research/jr/`.
6. Code and data for benchmarks. `http://www.cs.ucdavis.edu/~olsson/research/jr/papers/jrcoexceptsAppendix`.
7. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 578–608, May 2004.
8. A. W. Keen and R. A. Olsson. Exception handling during asynchronous method invocation. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing*, number 2400 in Lecture Notes in Computer Science, pages 656–660, Paderborn, Germany, August 2002. Springer–Verlag.
9. Message Passing Interface Forum. `http://www.mpi-forum.org/`.
10. .NET framework developer's guide. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconusingdelegates.asp`.
11. R. A. Olsson and A. W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer Academic Publishers, Inc., 2004. ISBN 1-4020-8085-9.