Mobile Process Resumption In Java Without Bytecode Rewriting

Matthew Sowders
University of Nevada, Las Vegas
Las Vegas, Nevada 89154
E-mail: sowders@unlv.nevada.edu

Jan Bækgaard Pedersen
University of Nevada Las Vegas,
Las Vegas, NV, 89154,
E-mail: matt.pedersen@unlv.edu

***Index Terms*—ProcessJ, Process-Oriented Programming, Mobile Processes**

***Abstract*—In this paper we describe an implementation of mobile processes with polymorphic interfaces in the ProcessJ language. ProcessJ is a process oriented language based on CSP and the π-calculus. In the paper we demonstrate the translation of ProcessJ to Java/JCSP and illustrate how to implement mobile processes with polymorphic interfaces without rewriting bytecode; this requires some clever code generation in Java since it does not support polymorphic interfaces.**

I. Introduction

In this paper we present a technique for implementing transparent mobile processes with polymorphic resumption interfaces for the ProcessJ language in Java/JCSP.

As part of the ProcessJ compiler, we have developed a code generation technique that allows a process to suspend, and subsequently resume, in the middle of a code block. When translating the ProcessJ code to Java/JCSP, a direct translation is not possible because Java does not support process suspension/resumption natively; nor does it support polymorphic interfaces.

Unlike previous attempts at resumable processes in Java [1] which describes a technique for implementing mobile processes with just one interface, the techniques described in this paper does not require any rewriting of the compiled Java bytecode. An integral part of the technique in [1] required the compiled bytecode to be changed to add explicit jumps to the resumption point. The technique described here does not need any such rewrite. Although no goto instruction is available in Java, we achieve resumption by using nested switch statements and collected state information.

Before describing the translation approach, we provide a brief overview of ProcessJ, mobility and resumability

A. ProcessJ

ProcessJ is a general purpose process-oriented programming language developed at the University of Nevada Las Vegas. The process-oriented primitives in ProcessJ are based on Communicating Sequential Processes, CSP [2], and the π-Calculus [3].

The syntax of ProcessJ is similar to that of Java. There are no classes or objects, but the expression syntax and control flow would be familiar to a Java programmer. Though the syntax is similar to Java, the semantics are similar to occam-π [4]. As a process-oriented language, ProcessJ is composed of *processes* that each execute in their own context similar to processes in occam-π [4].

```
 1:  mobile proc void foo(int x, int y) {
 2:      int a;
 3:      $B_1$
 4:      while ($B_2$) {
 5:          int q;
 6:          $B_3$
 7:          suspend resume with (int z);
 8:          int w;
 9:          $B_4$
10:      }
11:      $B_5$
12:  }
```

Fig. 1. Sample ProcessJ Code.

Figure 1 illustrates a fairly simple ProcessJ mobile process consisting of 2 interfaces. The *original* process interface, line 1, takes two integer parameters *x* and *y*. The second interface, a *resumption* interface, takes one integer parameter *z* at line 7. Furthermore, a number of local variable declarations (lines 2, 5, and 8) along with 5 code blocks (lines 3, 4, 6, 9, and 11) are included. The first time *foo* is called (we refer to that as *started*), it must be passed two integer values. When the *suspend* statement (line 7) is encountered, the process temporarily suspends, and control is returned to the caller. The caller can either re-invoke the process or transmit it to another process across a channel. When *foo* is invoked for the second time, it must be with the interface defined by the *suspend resume* statement (line 7), namely with just one integer value.

It should be fairly simple to imagine a translation of this code to Java for all the lines except the *suspend resume* statement in line 7. If Java had a goto statement, the implementation could be done using gotos and a some internal state (that is the end-result of the bytecode rewriting in [1]), but transforming the code into nested switch statements to achieve this as well as handling the different interfaces is more challenging.

B. Mobility

A classification of Mobile Code Languages is provided in [5]. The term *Strong Mobility* applies to an agent or process that is able to suspend their execution and be sent to a separate *computational environment* (CE) where it is resumed. The CE in the case of ProcessJ is either the same Java Virtual Machine, JVM, or it could mean a separate JVM. The resumed process is in the same execution state as when it was suspended.

In ProcessJ, we also offer *transparent strong mobility* [6]. The transparency comes from the ProcessJ programmer not needing to explicitly provide code to reestablish the state of the process.

C. Resumability

Pedersen and Kauke provide a definition of resumability in [1]. To summarize, a process is resumable if it contains a *suspend* statement. The *suspend* statement returns control the the caller, at some later point on the same or different JVM the process is resumed at the statement following the *suspend*. Since the publication of [1], the addition of polymorphic resumption interfaces, discussed later in section III, has added an additional statement *suspend with resume* that allows the process to be resumed with different parameters.

We need not bother with the specifics of bytecode definitions of resumability because our approach does not utilize bytecode rewriting.

It should also be noted, that the techniques described in this paper are equally well suited to mobile processes that do not have polymorphic interfaces, that is, implementing single interface mobile processes in Java can be achieved by using this approach as well.

II. Serializable Processes

A mobile process in ProcessJ is implemented as a Java class. Each mobile process class implements the Java Serializable interface. Implementing the Serializable interface allows us to transfer a process to another computational environment for resumption.

State is saved and restored by implementing processes as Serializable Java classes with all local variables rewritten as fields. Saving variables as fields is an easy and convenient way of preserving state between resumptions. The rewrite is accomplished by prefixing local variable names with a block id that makes them unique at the field level. The compiler is allowed to rewrite locals as fields because ProcessJ does not have any fields.

Storing variables as fields instead of locals is the crux that makes state restoration so simple. Rather than storing local state each time there is a suspend, state is stored every time a variable is mutated. There is also no need to restore variable values during resumption because they are already available.

III. Polymorphic Resumption Interfaces

ProcessJ supports polymorphic resumption interfaces [7]. That means, a process can be started with an interface A, execute, suspend, and later resume with a potentially different interface B. This is useful in combination with resources that are only available in the current computational environment, and when the use of 'dummy' parameters would otherwise be necessary.

```
mobile proc void foo(int x, int y, int z) {
    int a;
    $B_1$
    while ($B_2$) {
        int q;
        $B_3$
        suspend;
        int w;
        $B_4$
    }
    $B_5$
}
```

Fig. 2. Sample ProcessJ code with a single interface.

Polymorphic resumption interfaces allows the compiler to do static scope checking using interfaces that would otherwise require unused 'dummy' parameters by splitting a single interface into multiple interfaces. Consider Figure 2. If we were to use a single interface, but semantically we only use x, and y in block B_1 and only use w in block B_4 then we are expecting the caller to use two separate *implicit* interfaces while starting *foo*. It is their responsibility to know which variables are actually necessary.

Consider another situation where a process has two interfaces: the first is the reading end of a channel and the second is the reading end of a channel and the writing end of a channel. To implement this process with a single interface, the interface would have to be the super-set of the two interfaces: two channel reading ends and one writing end. There would be no clear distinction from the single interface when each is channel end is valid for use. For instance, if the first channel end is meant only for initialization, and the second and third are meant to be used as the process is passed around. In this case, you would first invoke the process with a valid first channel and with 'dummy' channels for the second and third. After the process suspends, you would then resume the process with a 'dummy' first channel end and valid second and third ends. What happens when the two are somehow transposed? In the best case the system becomes deadlocked and in the worst case the system is still able to communicate but acts in an unexpected manner. It is exactly this situation we are trying to avoid.

To implement polymorphic resumption interfaces, we use a single variadic function called *run*. A client calls *run* with the appropriate parameters for the current interface of the process. When a process resumes, it checks that the invoked interface and the current resumption interface are compatible.

In the translated Java code we use exceptions, which do not exist in ProcessJ, to indicate when an incorrect interface was used. During run time, the process checks that it is in the correct state for the current interface. The process throws

```
public class AbstractCSProcess implements CSProcess, Serializable {
    protected int control(int level) {...}
    protected <T> T getParameter(int index) {...}
    protected boolean isRunning(){...}
    protected void resume(Class<?>... parameterTypes){...}
    public void run(Object... args) {...}
    protected abstract void start();
    protected void suspend(int... targets) {...}
}
```

Fig. 3. Methods in AbstractCSProces

an *IncorrectInterfaceException* during run time if it is not in the correct state. This exception is a *RuntimeException* so it need not be checked. The benefit of throwing the *IncorrectInterfaceException* during run time is it allows a developer to know the interface expected, the interface sent, and the caller of the process when a programming error was made.

IV. Control Flow Rewriting

The general outline for ProcessJ code with a suspend is depicted in Figure 1 and the corresponding generated code is depicted in Figure 4. The example is a simple while statement that goes over the basic technique used to rewrite from ProcessJ to Java. Later in the section, we will give examples of how the process changes slightly with each of the other control structures.

Starting from the top and working our way down the code in Figure 4, we will explain each of the rewrites as they appear. The process is first converted into a Java class that extends *AbstractCSProcess*. The *AbstractCSProcess* is a base class that maintains state information and helper methods like *suspend*, *resume*, and *control* for navigating the control structure. The API is displayed in Figure 3

As mentioned previously, all mobile processes are serializable. Serializable processes will allow ProcessJ to send processes in a distributed environment when a distributed run time is available.

The next rewrite converts local variable definitions into fields. This removes the necessity of storing local state at suspend time and restoring it during a resume because everything is stored in fields. To avoid naming conflicts of variables in different scopes a compiler generated prefix is generated for each variable. For example, *a* was defined in block B_1 so it is renamed *$b1$a*.

At each control structure that uses an expression, a field is created to store the resulting value. In this case, B_2 is stored in *$c1*. Since an expression is only expected to be evaluated once, it cannot be re-evaluated during resumption. To address this, before the expression would normally be evaluated, the evaluated value is stored in a field corresponding to that control point.

The *start* method represents the body of the process. It begins by declaring each of the parameter variables. Again,

```
public class foo extends AbstractCSProcess {
    int $b1$a; // original a
    int $b3$q; // original q
    int $b4$w; // original w
    boolean $c1; // original B2

    @Override
    protected void start() {
        // interface 0 (int x, int y)
        int $i0$x, $i0$y;
        // interface 1 (int z)
        int $i1$z;

        switch (control(0)) {
        case 0:
            // interface 0 (int x, int y)
            resume(Integer.class, Integer.class);
            $i0$x = getParameter(0);
            $i0$y = getParameter(1);
            B1
        case 1:
            if (isRunning()) {
                $c1 = B2;
            }
            $1: while ($c1) {
                switch (control(1)) {
                case 0:
                    B3
                    suspend(1,1);
                    return;
                case 1:
                    //interface 1 (int z)
                    resume(Integer.class);
                    $i1$z = getParameter(0);
                    B4
                    $c1 = B2;
                } // end switch
            } // end while
            B5
        } // end switch
    } // end method start
} // end class foo
```

Fig. 4. General outline for generated Java code.

to avoid naming conflicts the compiler generates prefixes for each interface. For example, *x* is in interface 0 so it is renamed *$i0$x*. Next is the base switch statement of the body which splits B_1 and the *while* statement.

Inside case 0, we resume with a list of classes defined in *interface 0*. The *resume* method checks the provided interface is the same as the expected interface then sets the current state to *RUNNING* and resets the resume target. After that, we need to set the value of each parameter and execute B_1.

Java allows fall-through in switch statements. We use fall-through to split up the blocks without breaking the natural

```
...
$c1 = B₂;
continue $1;
...
```

Fig. 5. Example of a continue statement.

```
...
if (B₂) {
    B₃
    suspend resume with (int z);
    B₄
} else {
    ...
}
...
```

Fig. 6. Example if statement in ProcessJ.

control flow of the original program. After block B_1 executes, block B_2 needs to be evaluated but only if the process is currently running. If the process was suspended, the expression was already evaluated and stored in *$c1*.

Now we have reached the second level of control structure and B_3 is executed. The *suspend* method is then called with a control flow map to the next point of resumption. By 'control flow map', we mean a list of integers that describe the case statements that are selected to bring the process back to the next resumption point.

The process stores the control flow map and saves its state as *SUSPENDED* and returns control to the caller. The caller now has a reference to an object that can be serialized and saved, or sent over a channel, or immediately called again with the next interface.

When the process is resumed, the *control* method is called. The *control* method looks at the control flow map saved by the last suspend. It then jumps to case 1 and evaluates the previously stored value *$c1* at *while*. The *control* method then looks up the next level of control in the control flow map and again jumps to case 1.

The *resume* method checks the interface for the new *interface 1* and sets the state to *RUNNING*. The process then sets the value of the parameters and executes B_4. The expression B_2 is then re-evaluated and stored in *$c1* and the process continues to loop.

In the subsequent subsections we describe special circumstances for each of the Java control flow structures. In each of the examples you can replace the while statement of Figure 4 with the given generated code.

A. break

The *break* statement poses a small problem. In Java, *break* is used to terminate the enclosing *for*, *while*, *do-while* loop or *switch* statement. We are using *switch* statements to jump through blocks to resume points. If an unlabeled *break* is used in a block it would break out of the control flow *switch* instead of the original intended structure. We get around this by adding a label to each control structure and labeling unlabeled *break* statements.

```
...
if (isRunning()) {
    $c1 = B₂;
}
if ($c1) {
    switch(control(1)) {
    case 0:
        B₃
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i1$z = getParameter(0);
        B₄
    }
} else {
    ...
}
...
```

Fig. 7. Example of if statement with suspend.

B. continue

The *continue* statement skips the rest of the current iteration in a looping construct. Since a continue will skip updating the stored expression, the looping expression needs to be re-evaluated before a *continue* is executed. A *continue* is then rewritten as in Figure 5.

C. if else

The *if* statement needs almost no special treatment from that described above. An example of an *if* statement rewrite can be seen in Figure 7. The expression *$c1*, is stored as a boolean. The use of *$c1* ensures the expression is only evaluated while the process is in a *RUNNING* state.

There is no need to specify a label for the *if* statement. A *break* statement can be used to escape an *if* but it needs to specify a label to do so.

D. switch

The *switch* statement is rewritten similar to an *if* statement as seen in 9. The expression can be of type *byte*, *short*, *char*, and *int* [8], so the stored expression needs to be of the evaluated type.

Since *switch* statements allow fall-through from case to case, we need to consider the situation where B_2 evaluates to case 0, there is no *break* statement in case 0 and the control falls through to case1. In this situation, we need to update the stored value *$c1* at each case. Updating the stored value allows the process to resume to the last case statement the process was in.

Unlike the *if* statement, a *switch* needs a label. If an unlabeled *break* is within the *switch* statement, the *break* would inadvertently escape the generated control flow *switch* statement instead of the intended *switch*.

E. do-while

Figure 11 shows a *do-while* statement. Like other looping constructs and switch, *do-while* requires a label for *break* statements. Unlike other control structures, it is not necessary to store the result of the expression. Because the expression is evaluated at the end of the loop, it is never evaluated in the resumption process.

F. while

A *while* loop, was used in the main example in Figure 4. At the end of the loop, the expression is re-evaluated and stored.

```
...
$1:switch ($B_2$) {
case 0:
    $B_3$
    suspend resume with (int z);
    $B_4$
case 1:
    ...
default:
    ...
}
...
```

Fig. 8. Example of switch statement with suspend in ProcessJ.

```
...
if (isRunning()) {
    $c1 = $B_2$;
}
$1:switch($c1) {
case 0:
    $c1 = 0;
    switch (control(1)) {
    case 0:
        $B_3$
        suspend(1,1);
        return;
    case 1:
        // interface 1
        resume(Integer.class);
        $i1$z = getParameter(0);
        $B_4$
    }
case 1:
    $c1 = 1;
    ...
default:
    $c1 = 2; // not in other cases
    ...
}
...
```

Fig. 9. Example of switch statement with suspend.

```
...
$1: do {
    $B_3$
    suspend resume with (int z);
    $B_4$
} while($B_2$);
...
```

Fig. 10. Example of do-while loop with suspend in ProcessJ.

```
...
$1: do {
    switch(control(1)) {
    case 0:
        $B_3$
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i1$z = getParameter(0);
        $B_4$
    }
} while($B_2$);
...
```

Fig. 11. Example of do-while loop with suspend.

There is no need to check if the process is currently running at the end of a *while* because a suspended process will never reach this code.

The *while* statement also requires a label for *break* statements.

G. for

A *for* statement is best broken down into a *while* loop as seen in Figure 13. Before the loop, the expression and the initial value are both set only if the process in *RUNNING*. There is a label for *break* statements. At the end of the loop, the update is executed and the conditional expression is re-evaluated and stored.

Similar to the while statement, there is no need to check if the process is running because a suspended process will never reach this code.

H. Process

When a mobile process contains another mobile process, each process maintains its own state. For instance, if you invoke a mobile process *B*, from within a process *A*, and *B* suspends execution, that will only return control to *A*. The *A* process may decide to resume process *B*, or it may pass it down a channel.

It would also be possible for *A* to suspend and continue to hold a reference to process *B*. When *A* resumes it would also be possible to resume process *B* from *A* without any extra work other than what has already been explained in this paper.

```
...
for(init_expression; $B_2$; update_expression) {
    $B_3$
    suspend resume with (int z);
    $B_4$
}          ...
```

Fig. 12. Example of for loop with suspend in ProcessJ.

```
...
if (isRunning()) {
    init_expression;
    $c1 = $B_2$;
}
$1: while($c1) {
    switch(control(1)) {
    case 0:
        $B_3$
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i1$z = getParameter(0);
        $B_4$
    }
    update_expression;
    $c1 = $B_2$;
}
...
```

Fig. 13. Example of for loop with suspend.

V. Related Work

Mobile processes can be seen as a form of *process continuation* [9]. Unlike a traditional continuation, a process continuation represents the rest of a sub-computation from a given point in that sub-computation. Each process in ProcessJ executes in its own execution context. Therefore, a continuation of that process represents the control state of that one process, not the system as a whole.

An implementation of non-transparent weak mobility is available in Java through the use of jcsp.mobile [10], [11]. Though this implementation does allow process mobility, the end programmer needs to save all state and there is no way to save control state. One benefit to using jcsp.mobile is that it manages the class loading while communicating across JVMs.

Since ProcessJ already manages the state of the mobile process, jcsp.mobile may eventually find a place in the distributed run time. ProcessJ transparent strong mobility and jcsp.mobile's ability to manage dynamic class loading would be a strong combination.

Stefan Fünfrocken describes in [12] how to transparently migrate the state of a thread in Java. The approach described uses a preprocessor to instrument the Java code so no bytecode rewriting was necessary.

The difference between Fünfrocken's approach and ours is the need to migrate the entire thread stack. In ProcessJ, we are only interested in restoring the state of a single process, not the thread stack. Also, since we are able to convert all locals variables to fields, we need not save any state other than the current control state. This makes our approach much simpler for the implementation of ProcessJ.

Pedersen and Kauke describe in [1] how to provide transparent mobility in the JVM using a combination of code generation and bytecode rewriting. Since this paper is in large part an improvement on this work, lets look at how our implementation differs in greater detail. Our implementation is accomplished without bytecode rewriting, we use the new concept of polymorphic resumption interfaces, and there is no need to save local state before a suspend.

The bytecode rewriting adds an extra step to the flow. After Java code is produced, it must be compiled, then the bytecode rewritten before it can execute. In our approach, we produce Java code that is ready to compile and execute without modification.

In this approach we allow for polymorphic resumption interfaces where as [1] implements "resumability with parameter changes". Polymorphic resumption interfaces are a step above this because not only are the parameters allowed to change, but the interfaces changes as well.

One other difference lies in the points just before suspension and resumption. In [1] all state is stored in an activation record. The activation record is implemented as an array of objects on the process. Before a suspend, all the local state is saved into the activation record, and on resumption all local state is restored back to the proper local variables. Our implementation simplifies this drastically by moving all local variables into fields so there is no need to store values during suspension and restore during resumption.

VI. Conclusion

In this paper, we have shown how the ProcessJ compiler can provide transparent process mobility using only code generation. We have also shown how to implement polymorphic resumption interfaces and describe the rewriting steps required to provide these features in ProcessJ.

VII. Future Work

It is still necessary to perform the static scope checking proposed in [7]. This allows developers to know exactly where parameters are can be referenced.

All the necessary future work mentioned in [1] is still relevant. Handling channels inside mobile processes and other local resources still need to be resolved though the polymorphic resumption interfaces should help. Channels and other local resources in mobile processes could be handled by only allowing them in parameters and not stored as local variables. As mentioned previously, it would also be nice to use jcsp.mobile's dynamic class loading to load mobile processes across JVM.

To simplify matters a little, the code demonstrated is not yet integrated into JCSP. However, with little effort in the base class it should be possible to execute these processes as a *CSProcess*.

References

[1] J. B. Pedersen and B. Kauke, "Resumable Java Bytecode - Process Mobility for the JVM," in *The thirty-second Communicating Process Architectures Conference, CPA 2009, organised under the auspices of WoTUG, Eindhoven, The Netherlands, 1-6 November 2009*, 2009, pp. 159–172.

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978.

[3] R. Milner, *Communicating and mobile systems the pi-calculus*. Cambridge[England] ;;New York: Cambridge University Press, 1999.

[4] P. Welch and F. Barnes, "Communicating Mobile Processes: introducing occam-π," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. Springer Verlag, April 2005, pp. 175–210.

[5] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna, "Analyzing mobile code languages," in *Mobile Object Systems Towards the Programmable Internet*, ser. Lecture Notes in Computer Science, J. Vitek and C. Tschudin, Eds. Springer Berlin / Heidelberg, 1997, vol. 1222, pp. 91–109.

[6] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable support for transparent thread migration in java," in *Agent Systems, Mobile Agents, and Applications*, ser. Lecture Notes in Computer Science, D. Kotz and F. Mattern, Eds. Springer Berlin / Heidelberg, 2000, vol. 1882, pp. 377–426.

[7] J. B. Pedersen and M. Sowders, "Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces," in *The thirty-third Communicating Process Architectures Conference, CPA 2011, organised under the auspices of WoTUG, Limerick, Ireland, June 19-22 2011*, 2011, pp. 71–85.

[8] K. Arnold, J. Gosling, and D. Holmes, *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.

[9] R. Hieb and R. K. Dybvig, "Continuations and concurrency," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 128–136. [Online]. Available: http://doi.acm.org/10.1145/99163.99178

[10] K. Chalmers and J. M. Kerridge, "jcsp.mobile: A Package Enabling Mobile Processes and Channels," in *Communicating Process Architectures 2005*, sep 2005.

[11] K. Chalmers, J. M. Kerridge, and I. Romdhani, "Mobility in JCSP: New Mobile Channel and Mobile Process Models," in *Communicating Process Architectures 2007*, A. A. McEwan, W. Ifill, and P. H. Welch, Eds., jul 2007, pp. 163–182.

[12] S. Fünfrocken, "Transparent migration of java-based mobile agents," in *Mobile Agents*, ser. Lecture Notes in Computer Science, K. Rothermel and F. Hohl, Eds. Springer Berlin / Heidelberg, 1998, vol. 1477, pp. 26–37.