Static Scoping and Name Resolution for Mobile Processes with Varying Resumption Interfaces

UNLV School of Computer Science Technical Report (TRS-10-001)

Jan Bækgaard Pedersen and Matthew Sowders
University of Nevada Las Vegas
4505 Maryland Parkway
Las Vegas, NV, 89154
United States of America
Tel.: +1 702 895 2557; Fax: +1 702 895 2639
E-mail: matt.pedersen@unlv.edu, matthewsowders@gmail.com

Abstract

In this paper we consider a refinement of the concept of mobile processes in a process oriented language. More specifically, we investigate the possibility of allowing resumption of suspended mobile processes with different interfaces. This is a refinement of approach taken currently in languages like occam-π. The goal of this research is to implement varying resumption interfaces in ProcessJ, a process oriented language being developed at UNLV.

Introduction

In this paper we re-define static scoping rules for mobile processes with multiple (varying) suspend/resume interfaces, and develop an algorithm to perform correct name resolution.

One of the core ideas behind mobile processes is the ability to suspend execution (almost) anywhere in the code and return control to the caller, who can then treat the suspended process as piece of data, that can be transmitted to a different (physical) location, and at a later point in time, resumed and continue executing from where it left off.

In a language like occam-π [16], mobile processes are all initially invoked and subsequently resumed with the original (procedure) interface; that is, every resumption requires the same parameter list, even if some of these parameters have no meaning for the code that is to be executed. An example from [17] is as follows:

```
MOBILE PROC reindelf (CHAN AGENT.INITIALISE initialise?,
                      SHARED CHAN AGENT.MESSAGE report!,
                      SHARED CHAN INT santa.a!, santa.b!)
IMPLEMENTS AGENT
... local state declarations
SEQ
... in station compound (initialise local state)
```
WHILE TRUE
  SEQ
    ... in station compound
    SUSPEND -- move to gathering place
    ... in the gathering place
    SUSPEND -- move to santa’s grotto
    ... in santa’s grotto
    SUSPEND -- move to compound

The reindelf process only uses the initialise channel in the ... in station compound (initialise local state) code block. For each subsequent resumption of this process, a ‘dummy’ channel-end must be passed as the first actual parameter; a channel end representing a channel on which no communication is ever going to happen. Not only does that make the code harder to read, but also opens the possibility of incorrect code, should the channel be attempted used for communication in the subsequent code blocks. Similarly, should subsequent resumptions of the process require different channels, the initial call must provide ‘dummy’ values for these the first time the process is called.

For ProcessJ [13], a process oriented language being developed at the University of Nevada, Las Vegas, we propose a different approach to mobile process resumption.

When a process explicitly suspends, it defines with which interface it should be resumed. This of course means that parameters from the previous resumption are no longer valid. Static scoping analysis as we know it no longer suffices to perform name resolution; in this paper we present a new approach to name resolution for mobile processes with multiple resumption interfaces.

In ProcessJ, a suspend point is represented by the three keywords suspend resume with followed by a parameter list in parentheses (like a formal parameter list for a procedure as found in most languages). A suspended mobile process is resumed by a simple invocation using the name of the variable holding the reference to the suspended mobile, followed by a list of actual parameters (like a regular procedure call). For example, if a suspended mobile is held in a variable \( f \), and the interface defines one integer parameter, then \( f(42) \) is a valid resumption.

Let us start with a small example (no channels, but the example illustrates the problem):

```plaintext
mobile void foo(int x, int y) {
  int a;
  B_1
  while (B_2) {
    int q;
    B_3
    suspend resume with (int z);
  }
```
Figure 1: Simple ProcessJ example.

If we trace the code blocks executed and annotate them with the parameters from the most recently invoked interface we get:

$$B_1\{x, y\}, B_2\{x, y\}, B_3\{x, y\}, B_4\{z\}, B_5\{z\}, B_2\{z\}, B_3\{z\}, \ldots, B_2\{z\}, B_5\{x, y\} \text{ or } B_5\{z\}$$

If we look closely at this, we realize that the first time the code in block $B_3$ is reached the variables $x$ and $y$ can be referenced (they should hold values from the initial invocation of the process.) The second time the code block $B_3$ is executed will be during the second execution of the body of the while loops. This means that $foo$ has been suspended and resumed once, and since the interface of the suspend statement has just one variable, namely $z$, and not $x$ $y$, only $z$ may be referenced. So in general, we cannot guarantee that $x$ and $y$ can be referenced anywhere except block $B_1$. The same argument holds for $z$ in block $B_4$.

Table 1 shows in which blocks ($B_i$) the three interface variables can be referenced. Note, we have not considered the local variables $a$, $w$, and $z$ yet.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Blocks that may reference it</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$B_1$</td>
</tr>
<tr>
<td>$y$</td>
<td>$B_1$</td>
</tr>
<tr>
<td>$z$</td>
<td>$B_4$</td>
</tr>
</tbody>
</table>

Table 1: Parameters that can be referenced in various blocks.

If we had changed $z$ to $x$ (and retained their shared type int, all of a sudden, $x$ would now also be a valid reference in the blocks $B_2$, $B_3$, and $B_5$, that is, everywhere in the body of the procedure.

We start by examining the parameters of the interfaces, and later return to incorporate the local variables (for which almost regular static scoping rules apply), into a single name resolution pass containing both parameters and local variables.

In the next section we look at related work, and then proceed in section 2 to present a method for constructing a control flow graph (CFG) based on the ProcessJ source code. In section 3 we define sets of declarations to be used in the computation of valid reference, and in section 4 we illustrate how to compute these sets, and finally in section 5 we present the new name resolution algorithm for mobile processes with polymorphic interfaces.
1 Related Work

The idea of code mobility has been around for a long time. In 1969 Jeff Rulifson introduced a language called the Decode-Encode-Language (DEL) [15]. One could download a DEL program from a remote machine and the program would control communication and efficiently use limited bandwidth between the local and remote hosts [4].

Resumable processes are similar to mobile agents. [5] provides a classification of Mobile Code Languages (MCL). Strong Mobility is where the process code, state, and control state are saved before passing them to another process to resume at the same control state and with the same variable state. Weak Mobility in contrast does not preserve control state. Providing Mobility transparently means the programmer will not need to save the state before sending the process. All that is needed is to define the positions where the process can return control using a ’suspend’ statement or a ’suspend resume’ statement.

1.1 The Join Calculus and Chords

The Join Calculus [9] is a process algebra that extends Milner’s $\pi$-calculus [12] and that models distributed and mobile programming. Mobility is treated slightly different in the Join Calculus. Locality is inherent to the system and a process can define its locality rather than the suspend-send-resume approach used in occam-$\pi$.

$C_\omega$ [3] is a language implementation of the Join Calculus and an extension of the C# programming language. ProcessJ’s [13] approach to multiple resumption interfaces is different than the approach taken by $C_\omega$ which uses Chords. Chords are a method with multiple interfaces that can be invoked in any order, but the body of the method will not execute until every interface has been executed at least once. ProcessJ does not treat multiple interfaces this way; only one interface is the correct one at any time, and the process can only be resumed with that exact interface. Therefore, we are forced to either implement run-time errors, or allow querying the suspended mobile about which interface, it is ready to accept.

1.2 The Actor Model

ProcessJ also differs from Hewitt’s actor model [11, 10, 2] in the same way; In the actor model, any valid interface can be invoked, and the associated code will execute; again, for ProcessJ, only the interface that the suspended process is ready to accept can be invoked.

A modern example of the Actor Model is Erlang actors. Erlang uses pattern matching and \texttt{receive} to respond to messages sent. Figure 2 is a basic actor that takes several differing message types and acts according to each message sent. It is possible to specify a wild card ‘?’ message that will match all other messages so there is a defined default behavior. Erlang also has the ability to dynamically load
code on all nodes in a cluster using the nl command [1], or send a message to a process running on another node. A combination of these features could be used to implement a sort of weak mobility in Erlang.

```erlang
loop() ->
  receive
    'a' ->
      io:format('a'),
      loop();
    {Pid, 'b'} ->
      Pid ! 'echo',
      loop();
    _ ->
      io:format(do not know what to do'),
      loop()
  end.
```

Figure 2: Erlang Actors can respond to multiple message interfaces.

1.3 Delimited Continuations and Swarm

In 2009, Ian Clarke created a project called Swarm [6] that provides a framework for transparent scaling of distributed applications utilizing delimited continuations in Scala. A delimited continuation, also known as a functional continuation [8], is a continuation that is made into a function so it can return a value. This also has the added benefit of making them composable.

The goal of Swarm is to deploy an application to an environment with distributed data and move the computations to where the data resides instead of moving the data to where the process resides. This approach is similar to that used in MapReduce [7] though it is more broadly applicable because not every application can map to the MapReduce paradigm.

1.4 occam-π Versus ProcessJ Mobiles

The occam-π language has built in support for mobile processes [16]. The method adopted by occam-π allows processes to suspend rather than always needing to complete. A suspended process can then be sent down a channel and resumed from the same state it was suspended providing Strong Mobility. The interface defined is also slightly different than the previous versions of occam allowing the programmer to cleanly define a process and the interface it implements.

In occam-π, a mobile process must implement a mobile process type [16]; this is to assure that the process receiving the (suspended) mobile will have the correct
set of resources to re-animate the mobile. Mobile processes in ProcessJ with varying resumption interfaces cannot make use of such a technique, as there is no way of guaranteeing that the receiving process will re-animate the mobile with the correct interface. Naturally this can be rather detrimental to the further execution of the code; it would create a run-time error if the mobile were attempted executed. In ProcessJ we approach this problem (though not the scope of this paper, but worth mentioning) in the following way: It is possible to query a mobile process about its next interface (the one waiting to be invoked); this can be done in the following way:)

```java
MobileProc p = c.read();
if (p.accepts(chan<int>)) {
    chan<int> intChan;
    par {
        p(intChan.read);
        c.write(42);
    }
}
```

In [14], a process is defined that allows ProcessJ to provide strong transparent process mobility. The process involves Java code generation and Bytecode rewriting. The result provides a means of implementation for suspending and resuming a process in the JVM. The technique allows for the same resumption interface, but with some modification to the code generation, it might be possible to expand on their idea to allow for multiple resumption interfaces.

## 2 Control Flow Graphs and Rewriting Rules

The key idea to determine which variables can be referenced in various blocks of the body of a procedure is to determine which blocks in the program can be reached with the parameters of the different interfaces of the various suspend/resume points, and then compute intersections of these with respect to matching name/type pairs.

We will develop this technique through the example code in Fig. 1. The first step is to generate a source code-based control flow graph (CFG), which can be achieved using a number of simple graph construction rules for control diverting statements (these are `if`, `while`, `do`, `for`, `switch`, and `alt`-statements). These rules are illustrated in Fig. 3.
... if \((b)\) 
\(S_1\)  
else \(S_2\)  
...  

if-then-else statement.  

... while \((b)\) 
\(S\)  
...  

while statement.  

... for \((i; e; u)\) 
\(S\)  
...  

for statement.  

... switch \((e)\) {  
  case \(c_1\): \(B_1\)  
  ...  
  case \(c_n\): \(B_n\)  
}  
...  

switch statement.  

Figure 3: CFG construction rules.
If we apply the CFG construction rules from Fig. 3 in which we treat procedure calls and suspend/resume statements as non-control-diverting statements (The original process interface can be thought of as resume point and will thus be the first ‘statement’ in the first block in the CFG.), we get the control flow graph shown in Fig. 4.

Note, the I_0 before B_1 represents the original procedure interface, and the I_1 between B_3 and B_4 represents the suspend/resume interface.

Figure 4: CFG for the example code in Fig. 1.

Having the initial interface and the suspend/resume statements mixed with the regular block commands will not work for the analysis to come, so we need to separate those out. This can be done using a simple graph rewriting rule; each interface gets its own node. This rewriting rule can be seen in Fig 5.

We will refer to the nodes representing interfaces as interface nodes and all others (with code) as code nodes. With an interface node we associate a set of name/type/interface triples (n_i, t_i, I_i), namely the name (n_i) of the parameter, its type (t_i) and the interface (I_i) in which it was declared.

We introduce interface nodes for suspend/resume points into the graph in the following manner: if a code block B_i has m suspend/resume statements, then split B_i into m + 1 new code blocks B_{i1}, ..., B_{im+1} interspersed with interface nodes I_{i1}, ..., I_{im}.

Figure 5: CFG rewriting rule.

B_{i1} and/or B_{im+1} might be empty code nodes (Technically, so might all the
other code nodes, but that would be a little strange, as that would signify 2 or more suspend statements following each other without any code in between). Also, since the parameters of the procedure interface technically also make up an interface, we need to add an interface node for these as well. This is also covered by the rewriting rule in Fig. 5, and in this case $B_{i_1}$ will be empty and $I_{i_2}$ will be $I_0$. Rewriting the CFG from Fig. 4 results in the graph depicted in Fig. 6.

![Diagram of the altered CFG](image)

Figure 6: The altered CFG of the example in Fig. 4.

We now have a CFG with code and interface nodes. Each interface node has information about the parameters it declares, as well as their types.

### 3 In and Out Sets

Let us briefly consider the different types of node/arc configurations we can encounter in the transformed CFG. These are illustrated in Fig. 7.

For the nodes representing an interface, $I_i$, we are not interested in the incoming arcs. Since an suspend/resume point represented by an interface node re-defines which parameters can be accessed, they will overwrite any existing parameters. We
can now define, for each node in the CFG, sets representing incoming and outgoing parameters.

We define two sets for each node \( N \) (\( N \) is either a code node \( (B) \) or and interface node \( (I) \)) in the CFG, namely the in set \( (\mathcal{I}_k(N)) \) and the out set \( (\mathcal{O}_k(N)) \). Each of these sets are subscripted with a \( k \) denoting a 'generation' as we will be generating generations of these in an iterative fashion.

The in set of a code block ultimately represent the parameters that can be referenced in that block.

The out set for a code block is a copy of the in set; while technically not necessary, they make the algorithm that we will present later look nicer.

For interface nodes, in sets are ignored (there is no code in an interface node).

We can now define the following generation 0 sets for an interface node \( I_i \) and a code node \( B_i \):

\[
\begin{align*}
\mathcal{I}_0(I_i) & := \emptyset \\
\mathcal{O}_0(I_i) & := \{(n_{i,1}, t_{i,1}, I_i), \ldots, (n_{i,k}, t_{i,k}, I_i)\} \\
\mathcal{I}_0(B_i) & := \emptyset \\
\mathcal{O}_0(B_i) & := \emptyset
\end{align*}
\]

Since an interface node introduces a new set of parameters, we define its out set.

The \( k + 1 \)th generation of in and out sets can easily be computed based on the \( k \)th generation. Recall that a parameter (of a certain name and type) can only be referenced in a code block \( B \) if all interfaces \( I_i \) that have a path to \( B \) define it (both name and type must be the same!); this leads us to the following definition of the \( k + 1 \)th generation for in and out sets:

\[
\begin{align*}
\mathcal{I}_{k+1}(I_i) & := \emptyset \\
\mathcal{O}_{k+1}(I_i) & := \mathcal{O}_k(I_i) \\
\mathcal{I}_{k+1}(B_i) & := \bigcap_{(N,B_i) \in \text{CFG}} \mathcal{O}_k(N) \\
\mathcal{O}_{k+1}(B_i) & := \mathcal{I}_{k+1}(B_i)
\end{align*}
\]

That is, the parameters that can be referenced in a block is the intersection of the out sets of all its predecessors in the control flow graph.
To determine the set of references that are valid within a code block we repeatedly apply the four rules (only the two rules for the code blocks will change any sets after the first iteration) until no sets change.

Table 2 shows the results after two generations; the third does not change anything, so the result can be observed in the column labelled $I_1$.

<table>
<thead>
<tr>
<th></th>
<th>$I_0$</th>
<th>$O_0$</th>
<th>$I_1$</th>
<th>$O_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_0$</td>
<td>{}</td>
<td>${(x \text{ int } I_0), (y \text{ int } I_0)}$</td>
<td>{}</td>
<td>${(x \text{ int } I_0), (y \text{ int } I_0)}$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>{}</td>
<td>{}</td>
<td>$(x \text{ int } I_0), (y \text{ int } I_0)$</td>
<td>$(x \text{ int } I_0), (y \text{ int } I_0)$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>{}</td>
<td>{}</td>
<td>$(x \text{ int } I_0), (y \text{ int } I_0)$</td>
<td>$(x \text{ int } I_0), (y \text{ int } I_0)$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>$I_1$</td>
<td>{}</td>
<td>$(z \text{ int } I_1)$</td>
<td>{}</td>
<td>$(z \text{ int } I_1)$</td>
</tr>
<tr>
<td>$B_4$</td>
<td>{}</td>
<td>{}</td>
<td>$(z \text{ int } I_1)$</td>
<td>$(z \text{ int } I_1)$</td>
</tr>
<tr>
<td>$B_5$</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 2: Result of $in$ and $out$ sets after 2 generations.

To see that $x$ and $y$ or $z$ cannot be referenced in block $B_2$, consider the set $I_1(B_2)$:

$$I_1(B_2) := O_0(B_1) \cap O_0(B_4) = \{(x \text{ int } I_0), (y \text{ int } I_0)\} \cap \{(z \text{ int } I_1)\} = \{}$$

It is worth mentioning that when calculating the intersection, two triples $(n_i \ t_i \ I_i)$ and $(n_j \ t_j \ I_j)$ $(i \neq j)$ are compared only on the name $(n_i == n_j)$ and the type $(t_i == t_j)$. If they have the same name and type both triples will be represented in the result set (with different interface numbers of course.)

We can now formulate the algorithm for computing $in$ and $out$ sets.
4 Algorithm for In and Out Set Computation

Input: ProcessJ mobile procedure.

Method:

1. Using the CFG construction rules from Fig. 3, construct the control flow graph \( G \).

2. For each interface node \( I_i \), and code node \( B_i \) in \( G \) initialize
   \[
   \mathcal{I}_{k+1}(I_i) := \{ \} \\
   \mathcal{O}_{k+1}(I_i) := \mathcal{O}_k(I_i) \\
   \mathcal{I}_{k+1}(B_i) := \bigcap_{(N,B_i) \in CFG} \mathcal{O}_k(N) \\
   \mathcal{O}_{k+1}(B_i) := \mathcal{I}_{k+1}(B_i)
   \]

3. Execute this code:
   
   ```
   done = false;
   while (!done) {
     done = true;
     for (B ∈ G) do // only for code nodes
       \( B' = \bigcap_{(N,B_i) \in CFG} \mathcal{O}(N) \)
       if (\( B' \neq B \))
         done = false;
     \( \mathcal{O}(B) = \mathcal{I}(B) = B' \)
   }
   ```

Result: Input sets for all code block with valid parameter references.

It is worth pointing out that in the algorithm, generations of \( \text{in} \) and \( \text{out} \) sets are not used, this does not impact the correctness of the computation (recall, the operator used is the intersection operator), if anything, it shortens the runtime by allowing sets from generation \( k+1 \) to be used in the computation of other generation \( k+1 \) sets.

With this in hand, we can now turn to performing the actual scope resolution. This can be achieved using a regular static scope resolution algorithm with a small twist, as we shall see in the following section.

5 Static Name Resolution for Mobile Processes

As briefly mentioned in the previous section, the regular static name resolution algorithm works almost as-is. The only differences are that we have to incorporate the \( \text{in} \) sets computed by the algorithm in the previous section in the resolution pass,
and the way scopes are closed will differ slightly.

Different languages have different scoping rules, so let us briefly state the static scoping rules for parameters and locals in a procedure in ProcessJ.

- Variables cannot be re-declared in the same scope.
- A procedure declaration opens a scope in which only the parameters are held. The scope covers the entire body of the procedure.
- The body of a procedure opens a scope for locals declared at the top level of the body (this means that we can have parameters and locals named the same, but the parameters will be overshadowed by the locals.)
- The scope of a variable declared in the body of a procedure is the rest of the body from the declaration point.
- A block (a set of { }) opens a new scope (Variable names can now be reused, though re-declared variables overshadow other variables or parameters in enclosing scopes. The scope of a variable declared in a block is from the point of declaration to the end of the block.
- A for-statement opens a scope (it is legal to declare variables in the initialization part of a for-statement. The scope of such variables is the rest of the for-statement.
- A suspend/resume point open a new scope for the new parameters. Since we treat a suspend/resume point’s interface like the original procedure interface, an implicit block ensues immediately after, so a new scope is open for that as well (If we did not do this, we would break the rule that parameters and local can have shared names, as the in this situation would reside in the same scope.)

A symbol table in this context is a two dimensional table mapping names to attributes. In addition, a symbol table has a parent (table), and an access list of block numbers that represent which blocks may perform look-ups in them. If the use of a name in block $B_i$ requires a look-up in a table that does not list $i$ in its access list, the look-up query is passed to the parent recursively, until either the name is successfully resolved, or the end of the chain of tables is reached, and an error is returned.

Using the example from Fig. 1, a total of 5 scopes are opened, two by interfaces (The original procedure’s interface declaring parameters $x$ and $y$, accessible only by code in block $B_1$, and the suspend/resume point’s interface declaring parameter $w$ and $z$, accessible only by code in block $B_4$), one by the main body of the procedure (declaring local variable $a$), one by a block (declaring local variable $q$), and one following the suspend/resume point (declaring the local variable $z$, which
overshadows the parameter from the suspend/resume point).

In the code in Fig. 8, the code has been decorated with $+T_i$ to mark where the $i^{th}$ scope is opened, and $-T_i$ to mark where it is closed.

```java
mobile void foo$^{+T_0}$(int $x$, int $y$) implements mobileFooType {$^{+T_1}$
  int $a$;
  $B_1$
  while ($B_2$) {$^{+T_2}$
    int $q$;
    $B_3$
    suspend resume with $^{+T_3}$(int $z$); $^{+T_4}$
    int $w,z$;
    $B_4$
  }$^{-T_4,-T_3,-T_2}$
  $B_5$
}^{-T_1,-T_0}
```

Figure 8: Simple ProcessJ example annotated with scope information.

Note the closure of three scopes, $-T_4, -T_3, -T_2$, at the end of the block making up the body of the `while`-loop. Since there are no explicit markers in the code that closes down scopes for suspend/resume points ($T_3$), and the following scope ($T_4$), these get closed automatically when an enclosing scope ($T_2$) is closed. This is easily controlled when traversing the code (and not the CFG), as a typical name resolution pass would.

Figure 9 illustrates the 5 symbol tables, the symbols they declare, their access lists, and the nodes in the CFG with which they are associated.

We summarize in Table 3 which variables (locals and parameters) can be referenced in which blocks. Note, although block 4 appears in the access list in symbol table $T_3$ in Fig. 9 (and the parameter $z$ is in $O_1(B_4)$), the local variable $z$ in table $T_4$ overshadows the parameter.

<table>
<thead>
<tr>
<th>Block</th>
<th>Locals</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1$</td>
<td>$a \in T_1$</td>
<td>$x \in T_0, y \in T_0$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$a \in T_1$</td>
<td>$-$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$q \in T_2, a \in T_1$</td>
<td>$-$</td>
</tr>
<tr>
<td>$B_4$</td>
<td>$w \in T_4, z \in T_4, q \in T_2, a \in T_1$</td>
<td>$z \in T_4$</td>
</tr>
<tr>
<td>$B_5$</td>
<td>$a \in T_1$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Table 3: Final list of which variables/parameters can be access in which blocks.
6 Results

We have presented a simple algorithm that can be applied to create a control flow graph (CFG) on a source code level, and an algorithm to determine which procedure parameters and suspend/resume parameters can be referenced in the code of a mobile procedure.

Additionally, we presented a method for performing static scope resolution on a mobile procedure (mobile process) in a process oriented language like ProcessJ. This analysis obeys the standard static scoping rules for local variables and also takes into account the new rules introduced by making a procedure mobile (and thus resumable in the ‘middle of the code’, immediately after the point of exit (suspend point)).
7 Future Work

The ProcessJ compiler generates Java code using JCSP to implement CSP primitives like channels, processes and alternations. Additional work is required to do the scope checking proposed as well as mobile resumption. A possible implementation can follow the approach taken in [14], which unfortunately requires the generated (and compiled) bytecode to be rewritten; this involved reloading the bytecode and inserting new bytecode instructions, something that can be rather cumbersome. However, we do have a new approach, which does not require any bytecode rewriting at all, but that is outside the scope of this paper. We expect to be able to report on this in a different paper in the near future.

References


[9] Cédric Fournet and Georges Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In Gilles Barthe, Peter Dybjer, Luí̊s Pinto,


