Communicating Process Architectures 2011
P.H. Welch et al. (Eds.)
IOS Press, 2011

Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces

Jan Bækgaard PEDERSEN [1], Matthew SOWDERS

School of Computer Science, University of Nevada, Las Vegas

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 the 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.

Keywords. ProcessJ, process oriented programming, mobile processes, static name resolution

Introduction

In this paper we redefine static scoping rules for mobile processes with polymorphic (multiple possible 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 a 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.

We shall use the word *start* the first time a mobile procedure is executed/invoked, and *resume* for all subsequent executions/invocations. Let us illustrate the problem with an example from occam-π. In occam-π [16], mobile processes are all initially started 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 shown in Fig. 1.

The `reindelf` process only uses the `initialise` channel (line 1) in the `in station compound (initialise local state)` code block (line 7). For each subsequent resumption (lines 11, 13, and 15) of this process, a 'dummy' channel-end must be passed as the first parameter. The channel end represents 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 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.

[1]Corresponding Author: *Jan Bækgaard Pedersen, 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`.

```
MOBILE PROC reindelf (CHAN AGENT.INITIALIZE initialize?,
                      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
:
```

Figure 1. occam-π example.

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 polymorphic 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 it, 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 without any channels or local variables:

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

Figure 2. Simple ProcessJ example.

The first (and only) time B_1 is executed, it has access to the parameters *x* and *y* from the original interface (line 1). The first time B_2 is executed will be immediately after the execution of B_1. That is, following the execution of B_1, which had access to the parameters *x* and *y*. B_2 cannot access *x* or *y*, as we will see shortly. If B_2 evaluates to *true* the first time it is reached, the process will execute B_3 and suspend itself. B_4 will be executed when the process is resumed though the interface that declares the parameter *z* (line 5). The previous parameters *x* and *y* are now no longer valid. To realize why these parameters *should no longer be valid*, imagine they held channels to the previous local environment (the caller's environment) in which the process was executed, but in which it no longer resides; these channels can no longer be used, so it is imperative that the parameters holding references to them not be used again. Therefore, B_4 can only reference the *z* parameter, and not *x* and *y*. But what happens now when B_2 is reached a second time? *x* and *y* are no longer valid, but what about *z*? Naturally *z* cannot be referenced by B_2 either as the first time B_2 was reached, the process was started through the original interface and there was no *z* in that interface.

Furthermore, if we look closely at the code, we also realize that the first time the code in block B_3 is reached, just like B_2, the parameters from the latest process resumption (which here is also the first) would be *x* and *y*. The second time the code block B_3 is executed will be during the second execution of the body of the *while* loop. This means, that *foo* has been suspended and resumed once, and since the interface of the *suspend* statement has just one parameter, namely *z*, and not *x* and *y*, neither can 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.

We can illustrate this by creating a table with a trace of the program and by listing with which parameters the most recent resumption of the process happened. Table 1 shows a trace of the process where B_2 is evaluated three times, the first two times to *true*, and the last time to *false*. By inspecting Table 1, we see that both B_2 and B_3 can be reached with disjoint sets of parameters; therefore disallowing referenced to both x and y as well as z. B_5 could have appeared with the parameters x and y had B_2 evaluated to *false* the first time it was evaluated, thus we can draw the same conclusion for B_5 as we did for B_2 and B_3.

Table 1. Trace of sample execution.

Started/resumed interface	Block	Parameters from latest resumption	Remarks
$foo(x, y)$	—	—	*foo*(**int** *x*, **int** *y*)
	B_1	$\{x, y\}$	
	B_2	$\{x, y\}$	$B_2 = true$
	B_3	$\{x, y\}$	
$foo(z)$	—	—	**suspend resume with** (**int** *z*);
	B_4	$\{z\}$	
	B_2	$\{z\}$	$B_2 = true$
	B_3	$\{z\}$	
$foo(z)$	—	—	**suspend resume with** (**int** *z*);
	B_4	$\{z\}$	
	B_2	$\{z\}$	$B_2 = false$
	B_5	$\{z\}$	

Table 2 shows in which blocks (B_i) the three interface parameters can be referenced. Later on we shall add local variables to the code and redo the analysis.

Table 2. Parameters that can be referenced in various blocks.

Parameter	Blocks that may reference it
x	B_1
y	B_1
z	B_4

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 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. Finally we wrap up with a result section and some thoughts about future work.

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]. Though not exactly similar to how a ProcessJ process can be sent to different computational environments, DEL could be considered the beginning of mobile agents.

Resumable processes are similar to mobile agents. In [5], Chess et al. provides a classification of Mobile Code Languages. In a Mobile Code Language, a process can move from one computational environment to another. A computational environment is container of components, not necessarily a host. For example, two Java Virtual Machines running on the same host would be considered two different computational environments.

The term *Strong Mobility* [5] is used when 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 in a potentially different computational environment. The term *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. The process scheduling is also transparent to the end programmer because mobile processes are scheduled the same as normal processes.

1.1. The Join Calculus and Chords

The Join Calculus [9] is a process algebra that extends Milner's π-calculus [12] and that models distributed and mobile programming. Mobility is treated slightly different in the Join Calculus. The Join Calculus has the concept of Locality, or the computational environment [5] where the process is executed. Locality is inherent to the system and a process can define its locality rather than the suspend-send-resume approach used in occam-π.

Cω [3] is a language implementation of the Join Calculus and an extension of the C# programming language. Cω uses *chords*, a method with multiple interfaces that can be invoked in any order. The body of the method will not execute until every interface has been invoked at least once. ProcessJ does not treat multiple interfaces this way; only one interface is correct at a 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 Hewitts' actor model [2,10,11] 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 *receive* to respond to messages sent. Figure 3 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 type of weak mobility in Erlang; this is illustrated in Fig. 3.

```
loop () →
    receive
        % If I receive a string "a" print "a" to standard out
        "a" →
            io:format("a"),
            loop();
        % If I receive a process id and a string "b"
        % write "echo" to the given process id
        {Pid, "b"} →
            Pid ! "echo",
            loop();
        % handle any other message I might receive
        _ →
            io:format("do not know what to do."),
            loop();
end.
```

Figure 3. Erlang Actors can respond to multiple message interfaces.

1.3. Delimited Continuations and Swarm

In 2009, Ian Clarke created a project called Swarm [6]. Swarm is a framework for transparent scaling of distributed applications utilizing delimited continuations in Scala through the use of a Scala compiler plug-in. A delimited continuation, also known as a functional continuation [8], is a functional representation of the control state of a process.

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 the 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 communicated on a channel and resumed from the same state it was suspended, providing strong mobility.

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 polymorphic interfaces cannot make use of such a technique, as there is no way of guaranteeing that the receiving process will resume the mobile with the correct interface. Naturally, this can be rather detrimental to the further execution of the code; a runtime error would be generated if the mobile is not in a state to accept the interface with which is is resumed. The runtime check added by the compiler is inexpensive and is similar in use to an *ArrayOutOfBoundsException* in Java. 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 as illustrated in Fig. 4. If a process is not in a state, in

```
MobileProc p = c.read(); // Receive a mobile on channel c
if (p.accepts(chan<int>.read)) { // is p's interface (chan<int>.read) ?
    chan<int> intChan;
    par {
        p(intChan.read); // Resume p with a reading channel end
        c.write(42);
    }
}
```

Figure 4. Runtime check to determine if a process accepts a specific interface.

which it is capable of accepting a resumption with a certain interface, the check will evaluate to false, and no such resumption is performed. This kind of check is necessarily a runtime check.

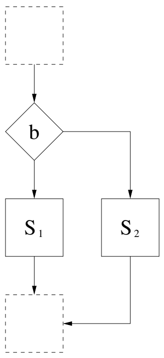
2. Control Flow Graphs and Rewriting Rules

The key idea to determine which parameters can be referred in a block, is to consider all paths from interfaces leading into that block. If all paths to a block include a definition from an interface of a parameter with the same name and type, then this parameter can be referenced in that block. This can be achieved by computing the intersection of all the parameters declared in interfaces that can flow into a block (directly or indirectly through other nodes.)

We will develop this technique through the example code in Fig. 2. 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 as well as *break* and *continue*). Theses rules are illustrated in Fig. 5.

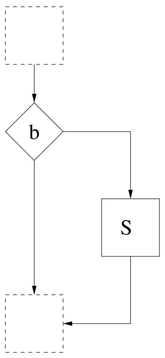
For the sake of completeness, it should be noted, that the depiction of the switch statement in Fig. 5 is based on each statement case having a *break* statement at its end; that is, there are no fall though cases. If for example B_1 could fall through to B_2 the graph would have an arc from e to B_1, from e to B_2, and to represent the fall through case, an arc from B_1 to B_2. *continue* statements in loops add an extra arc to the boolean expression controlling the loop, and a break in an *if* statement would skip the rest of the nodes from it to the end of the statement by adding an arc directly to the next node in the graph.

```
⋮
if ($b$)
    $S_1$
else
    $S_2$
⋮
```



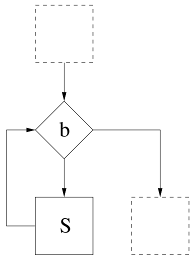
if-then-else statement.

```
⋮
if ($b$)
    $S$
⋮
```



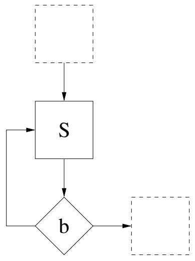
if-then statement.

```
⋮
while ($b$)
    $S$
⋮
```



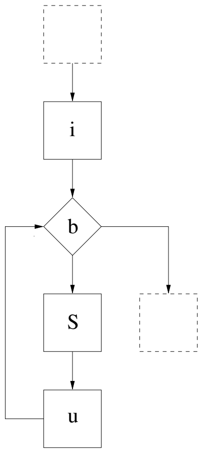
while statement.

```
⋮
do
    $S$
while ($b$)
⋮
```



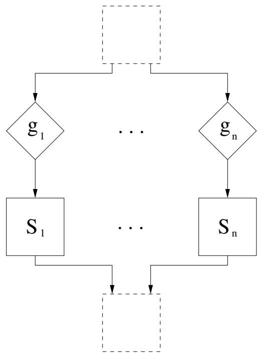
do statement.

```
⋮
for ($i$; $e$; $u$)
    $S$
⋮
```



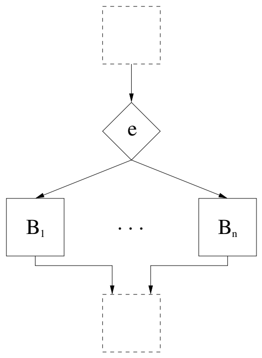
for statement.

```
⋮
alt {
    $g_1$:
        $S_1$
    ⋮
    $g_n$:
        $S_n$
}
⋮
```



alt statement.

```
⋮
switch ($e$) {
    case $c_1$: $B_1$
    ⋮
    case $c_n$: $B_n$
}
⋮
```



switch statement.

Figure 5. CFG construction rules.

If we apply the CFG construction rules from Fig. 5 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. 6

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*. Having the initial interface and the sus-

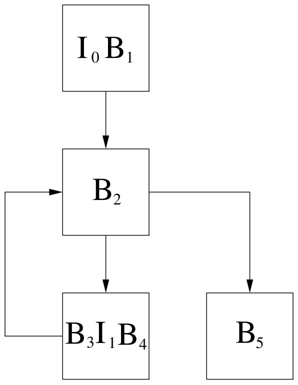


Figure 6. CFG for the example code in Fig. 2.

pend/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 is illustrated in Fig. 7.

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. In addition, we introduce a comparison operator $\hat{=}$ between triples defined in the following way: $(n_i\ t_i\ I_i) \hat{=} (n_j\ t_j\ I_j) \Leftrightarrow (n_i = n_j \wedge t_i = t_j)$. The corresponding set intersection operator is denoted $\hat{\cap}$. 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 $B_{i_1}, \ldots, B_{i_{m+1}}$ interspersed with interface nodes $I_{i_1}, \ldots, I_{i_m}$. B_{i_1} and/or $B_{i_{m+1}}$ might be empty code nodes (Technically, so might all the

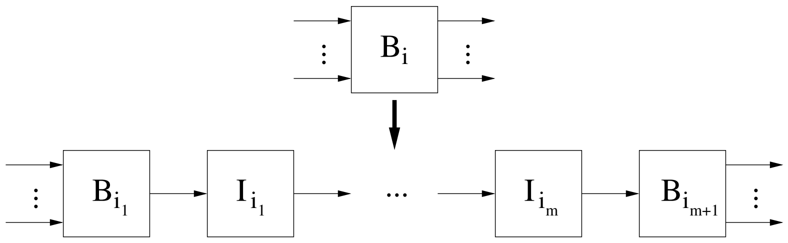


Figure 7. CFG rewriting rule.

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. 7, and in this case B_{i_1} *will* be empty and I_{i_2} will be I_0. Rewriting the CFG from Fig. 6 results in the graph depicted in Fig. 8. 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. This CFG is a directed graph (V_{CFG}, E_{CFG}), where the vertices in V are either interface nodes (I_i) or code nodes (B_i). An edge in E_{CFG} is a pair of nodes (N, M) representing a directed edge in the CFG from N to M; that is, if $(N, M) \in E_{CFG}$, then the control flows from the code represented by vertex N to the code represented by the vertex M in the program.

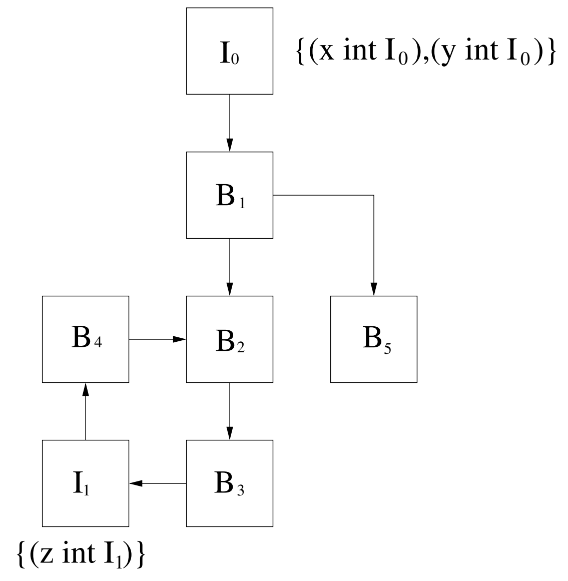


Figure 8. The altered CFG of the example in Fig. 6.

3. In and Out Sets

For the nodes representing an interface, I_i, we are not interested in the incoming arcs. Since a 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_i) or an interface node (I_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*. Generations of *in* and *out* set are dependent on the previous generations. 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 (representing an interface $(t_{i,1}\ n_{i,1}, \ldots, t_{i,k_i}\ n_{i,k_i})$) and a code node B_i:

$$\begin{array}{rcl} \mathcal{I}_0(I_i) & := & \{\,\} \\ \mathcal{O}_0(I_i) & := & \{(n_{i,1}\ t_{i,1}\ I_i), \ldots, (n_{i,k_i}\ t_{i,k_i}\ I_i)\} \\ \mathcal{I}_0(B_i) & := & \{\,\} \\ \mathcal{O}_0(B_i) & := & \{\,\} \end{array}$$

Since an interface node introduces a new set of parameters, we only 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_i if all interfaces I_j that have a path to Bi 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:

$$\begin{array}{rcl} \mathcal{I}_{k+1}(I_i) & := & \{\,\} \\ \mathcal{O}_{k+1}(I_i) & := & \mathcal{O}_k(I_i) \\ \mathcal{I}_{k+1}(B_i) & := & \hat{\bigcap}_{(N,B_i)\in E_{CFG}} \mathcal{O}_k(N) \\ \mathcal{O}_{k+1}(B_i) & := & \mathcal{I}_{k+1}(B_i) \end{array}$$

That is, the $k+1^{th}$ generation of the *in* set of block B_i is the intersection of the *out* sets of all its immediate predecessors at generation k in the CFG. 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 3 shows the results after two generations; the third does not change anything, so the result can be observed in the column labeled $\mathcal{I}_1$.

Table 3. Result of in and out sets after 2 generations.

	$\mathcal{I}_0$	$\mathcal{O}_0$	$\mathcal{I}_1$	$\mathcal{O}_1$
I_0	$\{\}$	$\{(x \text{ int } I_0), (y \text{ int } I_0)\}$	$\{\}$	$\{(x \text{ int } I_0), (y \text{ int } I_0)\}$
B_1	$\{\}$	$\{\}$	$\{(x \text{ int } I_0), (y \text{ int}\} I_0)\}$	$\{(x \text{ int } I_0), (y \text{ int } I_0)\}$
B_2	$\{\}$	$\{\}$	$\{\}$	$\{\}$
B_3	$\{\}$	$\{\}$	$\{\}$	$\{\}$
I_1	$\{\}$	$\{(z \text{ int } I_1)\}$	$\{\}$	$\{(z \text{ int } I_1)\}$
B_4	$\{\}$	$\{\}$	$\{(z \text{ int } I_1)\}$	$\{(z \text{ int } I_1)\}$
B_5	$\{\}$	$\{\}$	$\{\}$	$\{\}$

To see that x and y or z cannot be referenced in block B_2, consider the set $\mathcal{I}_1(B_2)$:

$$\mathcal{I}_1(B_2) := \mathcal{O}_0(B_1) \hat{\cap} \mathcal{O}_0(B_4) = \{(x \text{ int } I_0), (y \text{ int } I_0)\} \hat{\cap} \{(z \text{ int } I_1)\} = \{\ \}$$

If two triples have 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. 5, construct the control flow graph G.
2. For each interface node I_i, and code node B_j in $G = (V, E)$ initialize

$$\begin{aligned} \mathcal{I}_{k+1}(I_i) &:= \{\ \} \\ \mathcal{O}_{k+1}(I_i) &:= \mathcal{O}_k(I_i) \\ \mathcal{I}_{k+1}(B_j) &:= \hat{\bigcap}_{(N,B_j)\in E} \mathcal{O}_k(N) \\ \mathcal{O}_{k+1}(B_j) &:= \mathcal{I}_{k+1}(B_j) \end{aligned}$$

3. Execute this code:

```
done = false;
while (!done) {
    done = true;
    for ($B \in V$) do { // only for code nodes
        $B' = \hat{\bigcap}_{(N,B)\in E} \mathcal{O}(N)$
        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 in and out sets are not used. This does not impact the correctness of the computation (because 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

Let us re-introduce the code from Fig. 2, but this time with local variables added (lines 2, 5, and 8); this code can be found in Fig. 9. Also note, the local variable z in line 8 has the same name as the parameter in the interface in line 7. Naturally, this means that the interface parameter is hidden by the local variable.

```
 1:   mobile 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,z;
 9:           $B_4$
10:       }
11:       $B_5$
12:   }
```

Figure 9. Simple ProcessJ example with local variables.

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 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.

- Local variables cannot be re-declared in the same scope.
- An interface/procedure declaration opens a scope in which only the parameters are held. The scoping rules of interface parameters are what we defined in this paper.
- The body of a procedure opens a scope for local variables. (this means, that we can have parameters and locals named the same, but the parameters will be hidden by the local variables.)
- A block (a set of { }) opens a new scope (Local variable names can now be reused, though re-declared local variables hide other local variables or parameters in enclosing scopes. The scope of a local 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 opened 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. This access list contains the result of the algorithm that computed which blocks can access an interface's parameters. 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, resulting in an unsuccessful lookup of that name.

Using the example from Fig. 2, 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 hides the parameter from the interface of the suspend/resume statement).

In Fig. 10, the code has been decorated with $+\mathrm{T_i}$ to mark where the $\mathrm{i^{th}}$ scope is opened, and $-\mathrm{T_i}$ to mark where it is closed. Furthermore the implicit scopes opened by the parameter list of an interface, and the body following a suspend/resume statement have been added; these are the underlined brackets in lines 2, 12, 14, 17, 18, and 22.

Note the closure of three scopes, $-\mathrm{T}_4$, $-\mathrm{T}_3$, $-\mathrm{T}_2$, at the end of the block making up the body

```
 1:   mobile void foo
 2:   $\underline{\{}^{+\mathrm{T}_0}$
 3:       (int x, int y)
 4:       $\{^{+\mathrm{T}_1}$
 5:           int a;
 6:           $B_1$
 7:           while ($B_2$)
 8:           $\{^{+\mathrm{T}_2}$
 9:               int q;
10:               $B_3$
11:               suspend resume with
12:               $\underline{\{}^{+\mathrm{T}_3}$
13:                   (int z);
14:                   $\underline{\{}^{+\mathrm{T}_4}$
15:                       int w,z;
16:                       $B_4$
17:                   $\underline{\}}^{-\mathrm{T}_4}$
18:               $\underline{\}}^{-\mathrm{T}_3}$
19:           $\}^{-\mathrm{T}_2}$
20:           $B_5$
21:       $\}^{-\mathrm{T}_1}$
22:   $\underline{\}}^{+\mathrm{T}_0}$
```

Figure 10. Simple ProcessJ example annotated with scope information.

of the **while**-loop. Since there are no explicit markers in the code that close 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 11 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 4 which

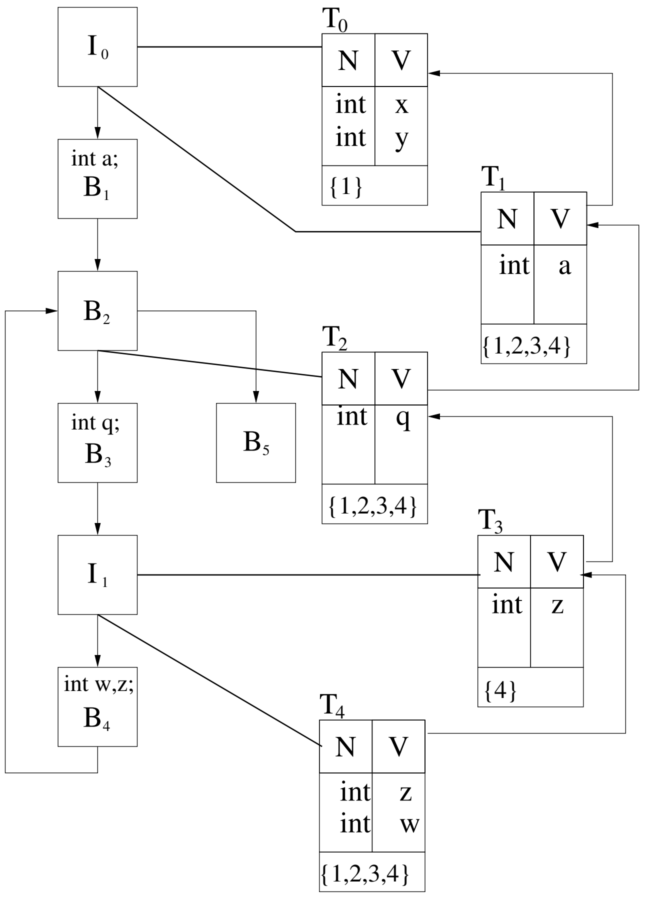


Figure 11. CFG with symbol tables.

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. 11 (and the parameter z is in $\mathcal{O}_1(B_4)$), the local variable z in table T_4 hides the parameter.

Table 4. Final list of which variables/parameters can be access in which blocks.

Block	Locals	Parameters
B_1	$a \in \mathrm{T}_1$	$x \in \mathrm{T}_0,\ y \in \mathrm{T}_0$
B_2	$a \in \mathrm{T}_1$	–
B_3	$q \in \mathrm{T}_2, a \in \mathrm{T}_1$	–
B_4	$w \in \mathrm{T}_4, z \in \mathrm{T}_4,\ q \in \mathrm{T}_2,\ a \in \mathrm{T}_1$	$z \in \mathrm{T}_3$
B_5	$a \in \mathrm{T}_1$	–

6. Results & Conclusion

We have presented an algorithm that can be applied to create a control flow graph (CFG) at 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 with polymorphic interfaces (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 implementation work is required to integrate the algorithm as well as the JCSP code generation into the ProcessJ compiler. A possible implementation of mobiles using Java/JCSP 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. We expect to be able to report on this in a different paper in the very near future.

References

[1] Ericsson AB. Erlang STDLIB, 2010. `http://www.erlang.org/doc/apps/stdlib/stdlib.pdf`.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, 1986.

[3] Nick Benton, Luca Cardelli, and Cedric Fournet. Modern Concurrency Abstractions for C#. In *ACM TRANS. PROGRAM. LANG. SYST*, pages 415–440. Springer, 2002.

[4] Peter Braun and Wilhelm Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[5] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea?Mobile Agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer Verlag, Berlin, 1997.

[6] Ian Clarke. swarm-dpl - A transparent scalable distributed programming language, 2008. `http://code.google.com/p/swarm-dpl/`.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[8] Matthias Felleisen. *Beyond continuations*. Computer Science Dept. Indiana University Bloomington, Bloomington IN, 1987.

[9] Cédric Fournet and Georges Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer Verlag Berlin / Heidelberg, 2000.

[10] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323364, June 1977.

[11] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. Actor induction and meta-evaluation. In *In ACM Symposium on Principles of Programming Languages*, pages 153–168, 1973.

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

[13] Jan B. Pedersen et al. The ProcessJ homepage, 2011. `http://processj.cs.unlv.edu`.

[14] Jan B. Pedersen and Brian 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*, pages 159–172, 2009.

[15] Jeff Rulifson. DEL, 1969. `http://www.ietf.org/rfc/rfc0005.txt`.

[16] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes: introducing occam-π. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[17] Peter H. Welch and Jan B. Pedersen. Santa Claus - with Mobile Reindeer and Elves. In *Fringe Presentation at Communicating Process Architectures conference*, September 2008.

Appendix

To illustrate the construction of the CFG in more depth, Fig. 13 shows the control flow graph for a *for* loop with conditional *break* and *continue*. The code from which the CFG in Fig. 13 was generated is shown in Fig. 12. In Fig. 13 the body of the *for* loop is represented by the largest shaded box, the *if* statement containing the *break* statement is the box shaded with vertical lines, and the *if* statement containing the *continue* statement is the box shaded with horizontal lines.

```
for ( $i$ ; $b_1$ ; $u$ ) {
    $B_1$
    if ($b_2$) {
        $B_2$
        break;
    }
    $B_3$
    if ($b_3$) {
        $B_4$
        continue;
    }
    $B_5$
}
```

Figure 12. Example code with conditional *break* and *continue* statements.

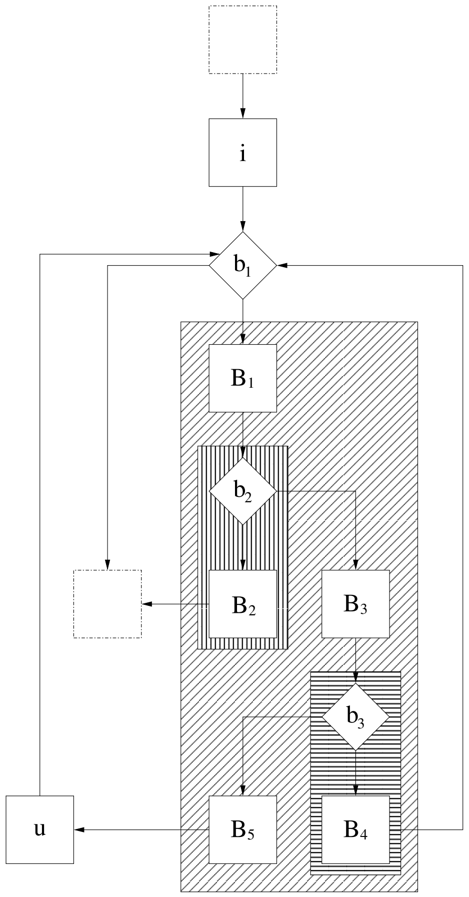


Figure 13. CFG for the example code shown in Fig. 12.