π-Calculus semantics of a Smalltalk like language.

Research paper by:

Else Nordhagen
Department of Informatics, University of Oslo
Pb. 1080, Blindern, 0316 Oslo, Norway
Phone: +47-2-45 37 13
Fax: +47-2-45 34 01
Email: lc@ifi.uio.no

Abstract:

The work presented here is done in order to compare traditional denotational semantics and π-calculus [Mil2/3] as tools for formally expressing the semantics of object-oriented languages. The target language for the comparison is the "pure" object-oriented language Smalltalk-80 which semantics is denotationally defined in [Kam] and [Wol].
The presented work shows that it is rather simple to express the denotation of an pure object-oriented language in the π-calculus. It also shows that using π-calculus simplifies the description of expressions which change the execution sequence. π-calculus also facilitates expressing the semantics of hybrid parallel object-oriented languages as shown for two parallel object oriented languages in [Walk] something which pushes the limit on what can be done with traditional denotational semantics.
The work presented may also form the basis for the study of object types by the equivalence relations defined for π-expressions.

π-Calculus semantics of a Smalltalk like language.

by Else Nordhagen

1 Introduction

The work presented here is done in order to compare denotational semantics and π-calculus as tools for expressing the semantics of object-oriented languages. It may also form the basis for the study of object types by the equivalence relations defined for π-expressions.

Several attempts have been done at defining the semantics of object-oriented languages. Traditionally this have been done using denotational semantics. [Kam] and [Wol] use this approach when defining semantics of the archetypical object-oriented language Smalltalk-80 [Gold]. The semantics of object-oriented languages have also been expressed using π-calculus [Mil2/3] as described in [Walk].

The language semantics presented here is built in the same way as described in [Walk], but the target language is a more "pure" object-oriented language; a subset of Smalltalk-80. For the purpose of this paper the subset is named Chat to avoid mixing it up with the full language. [Walk] does not handle inheritance, return statements and block-objects, something which is included here. The languages of [Walk] handles loops and branching more traditionally while this is here handled as messages to booelan objects. This description also handles description of how illegal messages are handled, something which [Walk] totally lack.

The Chat language will be introduced in steps, presenting more and more expressions. The first chapter below defines a very simple language which does not include Smalltalk features like boolean, symbol and block objects and does not have inheritance or a return-statement. In chapter 4 the semantics of this language is formally defined by using the π-calculus. In later chapters this language will be expanded to have more of these features:

Chapter 5 - inheritance added
Chapter 6 - return expression added
Chapter 7 - block-objects added

The main Smalltalk features which are not included in Chat are:

- Classes and methods as objects and method addition during runtime
- class and pool variables
- literal objects: string, number etc.
- selectors as objects (Symbols) and the "perform: primitive"
- Arrays and collections
- "cascading messages"; i.e. expressions of the form:

  Receiver MessageExpression1; MessageExpression2;... .

  where the Receiver will receive all the initial messages in the message expressions.
  It may alternatively be written:

  Receiver MessageExpression1. Receiver MessageExpression2.....

2 Definition of a simple object oriented language

2.1 Extended BNF definition

BNF has bee expanded with the following notation:

[ ]	meta-parentheses	M^*	zero or more Ms
M^+	one or more Ms	$M^?$	zero or one M
$M^*_;$	zero or more Ms separated by ; - * may be replaced by + or ?, and ; with any character		

Since Classes are not objects and therefore class and method creation is not done by message-passing, syntax must be provided to define programs, classes and methods:

Prog ≡ **prog declarations** ClassDecls **globals** Ident*; **vars** Ident*; ExprList **endProg**

ClassDecls ≡ [**class** Ident [**instVars** $\text{Ident}^+_,$]$^?_;$ MethodDecl]$^+_;$

MethodDecl ≡ [**method** Ident [**params** $\text{Ident}^+_,$]$^?_;$ [**vars** $\text{Ident}^+_,$]$^?_;$ $\text{Expression}^*_.$]$^+_;$

The rest of the syntax is "traditional" Smalltalk-80:

Expression ≡ ObjectReference | Assignment | MessageSend | ObjectCreation

ObjectReference ≡ Ident | **self**
Assignment ≡ Ident := Expression
MessageSend ≡ Expression Message^+
    Message ≡ UnarySelector | BinarySelector Expression | KeyWordMessage
    UnarySelector ≡ Ident
    BinarySelector ≡ + | - | * | / | = | < | > | etc.....
    KeyWordSelector ≡ [Keyword Expression]*
        Keyword ≡ Ident :
ObjectCreation ≡ Ident **new**

Note that the pseudo variable ´super´ is missing in the language. It is not added before inheritance is introduced in chapter 5.

2.2 Informal Semantics

In object creation the only legal Idents are those of classes. Variables within a class must have unique names and so must methods.

A keyword message has parameters separated by keywords which are idents ending with colon (:). The name of the message is the sequence of keywords.

Variables are always initialised to point at an object representing the "undefined object" named nil.
self is a pseudo-variable referring to the object where the expression occurs.

Some simple examples of expressions, TwoDPoint is Ident of a class, point, x, delta and otherPoint are Idents of variables and 10 and 20 are Idents of variables referencing objects presumably representing integer values:

```
point := TwoDPoint new          create a new object, and let point reference this object
```

x := `point x`	assign to x the return value of the unary message to point, name of message: "x"
`point x: 10 y: 20`	keyword message to point, message name: "x:y:"
`self` `move: otherPoint`	keyword message "move:" to the object executing this expression, otherPoint is parameter
`delta :=` `point x + 10`	first send "x" to point, and then send the binary message "+" with 10 as a parameter to the object returned form the "x"-message. Finally let delta reference the object returned from the "+"-message.

A messages is received by an object as a result of a message send expression. The receiving object is the object which reference is the value of the initial expression in the message send expression. On reception of a message the receiving object selects a method to execute. This selection is done by selecting the method with the same name as the message. In Smalltalk-80 a situation where there is no appropriate method in a receiving object is expressed as the receiver not understanding a message. In such an event a special message will be sent the receiving object by the underlying runtime system. This message has the name "messageNotUnderstood". This runtime behaviour is described below for Chat in the translation of the class-construct. In Smalltalk-80 the default method for "messageNotUnderstood" starts the debugger. Other object may perhaps have a method-implementation for "messageNotUnderstood" which pass the not understood message on to some other object. The "messageNotUnderstood" method is not defined below as it is not part of the language definition but made by the Smalltalk (Chat) programmer.

By having key-word messages and initialise variables to refer to an actual object which represent an empty pointer (named nil), the only kind of runtime error which can occur in Smalltalk/Chat is the sending of a message which is not understood by the receiver. This is because:

- the wrong number of parameters can not occur,
- the wrong type of parameter can only result in the sending of a message the receiver (the parameter with the wrong type) will not understand
- sending a message to something which is not an object will never occur since all variables "at least" point to the nil-object initially assigned to it and thus assuring that a variable always holds a reference to an object.
- a nil object will not understand messages which are meant for the "actual" objects the variable is supposed to refer to if properly initialised.

This message not understood runtime-error is handled within the running Smalltalk program itself in the methods for "messageNotUndersood", and not by some underlying run-time mechanism or by the compiler.

The only errors in the code of the program which are not handled by this mechanism is the use of undeclared variable and class names and declarations with non-unique names where they should be unique. These errors should be caught by the compiler. They are not handled in the semantics definition below.

3 Short description of the π-calculus

Below is given a short summary of the π-calculus expressions used in the semantic definitions[1]. For a more detailed description see [Mil2/3]. In π-calculus "overscore" is used, instead of as below "underscore". This is due to deficiencies in my text editor, and can be corrected in any final version of this paper

The π-calculus is a process calculus in which processes with changing communication structure may be expressed. The processes share communication channels and such channels can be passed on to other

---

[1] In π-calculus "overscore" is used, instead of as below "underscore". This is due to deficiencies in my text editor, and can be corrected in any final version of this paper

processes. Each channel has a name and the only "values" which may be communicated are such names. This means that in the π-calculus "variables", "communication channels" and "values" are all names. An infinite set N of names is presupposed, and in the below description single letter such as x,y,v (possibly with subscripts) range over names. Below the informal semantics of π will be expressed using words such as channel, parameter etc. to reflect the role the name has in the expression.

The basic building block of π-expressions are process expressions. P,Q range over such expressions which are built from the following expressions:

P ::=

$\underline{x}y.P$	output action: send name y on channel x
x(y).P	input action: receive an unknown name (say v) on channel x, and then behave like P{v/y) (P with v for y, v must be a new name not occurring in P)
(y)P	y is a private name for P, making it unique within the total system. y may be passed to other processes so that they can communicate on this channel.
[x=y]P	behave like P if name x is equal to name y
P | Q	behave as if P and Q act independently. P and Q may share channels and communicate on these.
P + Q	behave either like P of like Q
0	terminate (usually left out at end of expressions)

α is used to denote a sequence of input and output actions. The expressions α*P is defined as:

α*P == α.(P | α*P) Do α and spawn off a new similar process before continuing

$\underline{x}(y)$ is short for (y) $\underline{x}y$, that is send a new local name out on x.

4 π-Calculus semantics

The signature of the semantic functions is as follows:

[[Chat expression]] == π-expression

Functions are introduced to simplify expressing the semantics. New functions are defined right below the semantic definitions they are called.

In the below ExprList is used to denote a list of expressions separated by period (.).

In the following definition it is presumed that there is a global variable named #nil pointing to an object representing the undefined object pointer. w is the in channel to the object "executing" the expressions (representing the objects unique id) and v is the channel to return any value over.

Variable declaration:	[[instVar X_1,.....X_n]] == Decl(X_1,.....X_n)
	where Decl(X_1,.....X_n) == Reg_{x_1}(#nil) | ..... | Reg_{x_n}(#nil)
	and $Reg_x(y) == \underline{r_x}y.Reg_x(y) + w_x(z).\ Reg_x(z)$
Object reference :	[[X]] (w, v) == $r_x(z).\ \underline{v}z$
	[[self]] (w,v) == $\underline{v}w$
Assignment :	[[X:=E]] (w,v) == (z) ([[E]](w,z) | $z(u).\underline{w_x}u.\ \underline{v}u$)
Object creation :	[[C new]] (w,v) == $c(z).\ \underline{v}z$.
	c is here the channel from the agent representing the class of C.

UnaryMessage expression:

[[E M]] (w,v) == (v') ([[E]](w, v') | (u) $v'(w_2).\ \underline{w_2}u.\underline{u}m.u(m').\underline{m'}v$)

BinaryMessage expression :

[[E M E1]](w, v) == (v') ([[E]](w, v')
| (v1) (v'(w_2). [[E1]](w, v1)
| (u) (v1(p). $\underline{w_2}$u.$\underline{u}$m.u(m').$\underline{m'}$v.$\underline{m'}$p)))

KeywordMessage expression:

[[E M E1...En]](w, v) == ((v') ([[E]](w, v')
| (v_1) (v'(w_2). [[E1]](w, v_1)
| (v_2 p_1) (v_1(p_1) [[E2]](w,v_2)
| (v_3 p_2) (v_2(p_2) [[E3....
….
| (v_n p_{n-1})(v_{n-2}(p_{n-1})[[En]](w, v_n)
| (u p_n m') v_n(p_n).$\underline{w_2}$u.$\underline{u}$m.u(m').$\underline{m'}$v.$\underline{m'}$$p_1$...$\underline{m'}$$p_n$))))

M is here the keyword message name.

In these message-sending π-descriptions the expression given to the receiver is first handled. Then the parameter expressions are evaluated and finally the message is sent. Sending a message is done as follows:

- send a local channel name to the receiver = $\underline{w_2}$u
- send over this channel the message selector = $\underline{u}$m
- get back a channel to the method process = u(m')
- send return channel (v) to the method process = $\underline{m'}$v
- send parameters to the method process = $\underline{m'}$$p_1$...$\underline{m'}$$p_n$

Expression list :

[[E1. ... En]] (w,v) ==
(e v_1)([[E1]](w,v_1) | (v_2)(v_1(e).[[E2]](w,v_2) | ... | v_{n-1}(e).[[En]](w,v))))

The return value from the first n-1 expressions are not used. The empty message list is just return of the objects w-channel:

[[<empty>]] (w,v) == $\underline{v}$w

[[method M params P_1...P_m vars X_1...X_n; ExprList]] (m) ==
m(z).* (N) (Decl(P_1...P_m, X_1...X_n) | z(w).z(r).z(p_1).$\underline{w}_{p_1}$$p_1$...z($p_m$).$\underline{w}_{p_m}$$p_m$[[ExprList]](w,r))
with N = {p_i | p_i ε p_1...p_m } U {x_i | x_i ε x_1...x_n }

A method is turned into a process which first sends out a local channel. Then it spawns of a new process which is to be a new instance of the method to be contacted when a new call is done. A new method instance is created as the operator * is defined as follows:

α * P == α.(P | α * P)

and α = m(z) making z is a local channel for each instantiation.
The old process then continues with receiving the channel to the object this method is invoked on (w) and the channel to return the value on (r). Then the parameters P_1...P_m are received and stored. Finally the method body is executed.

[[MD_1...MD_n]] (m_1...m_n) == [[MD_1]](m_1) | ... | [[MD_n]](m_n)

[[class C instVars X_1...X_n; MD_1...MD_m]] (c) == $\underline{c}$(w). Body(w)
where Body (w) == * (N)(Decl(X_1...X_n) | [[MD_1...MD_m]] (M) | Answer(w))
with N == { r_x,w_x | X ε X_1...X_n } U M, M = { m | m ε MD_1...MD_m }
and Answer(w) == w(u) * u: [m_i ⇒ m_i(z) $\underline{z}$w.$\underline{u}$z,

[OTHER ⇒ (z) wz.z#messageNotUnderstood)]

and [m_i ⇒ ...] for all m_i ε M

and #messageNotUnderstood is the special message described above sent when receiving unknown messages.

[[$CD_1...CD_n$]] ($c_{1...}c_n$) == [[CD_1]](c_1) | ... | [[CD_n]] (c_n)

[[prog declarations $CD_1...CD_m$ globals $X_1...X_n$; vars $Y_1...Y_k$: ExprList endProg]] ==

(GUC) Decl($X_1...X_n$) | [[$CD_1...CD_m$]] (C) | (L, w, v) Decl($Y_1...Y_k$) | [[ExprList]] (w,v).v(x)

with G == { r_x,w_x | X ε $X_1...X_n$ }, C = {$c_{1...}c_m$}, L == { r_y,w_y | Y ε $Y_1...Y_k$ }

For each method name M_i in the program we assume a constant name m_i of the π-calculus, and for each class name C_i a constant c_i. When executing the translation functions from Chat to π-expression must have available the mappings from C -> c, M -> m and X -> r_x,w_x in order to find the right thing to translate into. One of the Y_i should be 'nil' which should be initialized to an instance of a class (one of the CDs) for undefined objects in the ExprList of the program.

5 Introducing inheritance into the language

To expand Chat to also include inheritance the following additions have to be made to the language:

ClassDecls = ...| class Ident super Ident instVars Ident*; MethodDecl

ObjectReference = ... | super

super may only be used for expressing message receiver.

Only single inheritance is described, but multiple inheritance could easily be done.

Inheritance is modelled by letting one object be represented by several objects, one object for each of the classes in the sub/superclass structure. Each superobjects have a channel to the basic object. This channel is named a. The class agent have to have features to create objects which are parts of other objects, there fore every class has a corresponding c and sc channel which they send "new" signals on, one for making basic objects (c) and one for making superobjects (sc). It is presumed that sub/superclasses have disjoint variable names.

[[class C instVars $X_1...X_n$; $MD_1...MD_m$]] (c) == c(w)*Body(w,w) | sc(w).w(a)*Body(w,a)

where a is a special channel to the basic object this object is part of

and Body(w,a) ==(N) (Decl($X_1...X_n$) | [[$MD_1...MD_m$]] (M) | Answer(w,a))

and Answer(w,a) == w(u) * u(m).([m=#asSuper] u(m).u(u)).

([m=m_i] m_i(z) za.uz +

[m=#super] u#nil +

[m=#self] ua +

[m=OTHER] (z) az.z#messageNotUnderstood)

[[class C super SC instVars $X_1...X_n$; $MD_1...MD_m$]] (c) == c(w)*SBody(w,w) | (a) sc(w).w(a)*SBody(w,a)

where SBody(w,a) == (N, s)(Decl($X_1...X_n$) | [[$MD_1...MD_m$]] (M) | sc(s).sa. SAnswer(w,a))

and SAnswer(w,a) == <same as Answer(w,a) except the two lines: >

[m=#super] us +

[m=OTHER] (z) sz.z#asSuper.zm.zu)

When a method is not found the message is passed on to the super object in stead of sending the error message. The extra new expression 'super' must also be defined:

[[super]] (w,v) == (u) wu. u#super.u(s). vs

self must also be modified to return the channel to the bottom object in the sub/super structure:

[[self]] (w,v) == $\underline{w}u.\underline{u}\#self.u(a).\underline{v}a$

When executing the translation functions from Chat to π-expression must also have available the mapping from SC -> sc.

6 Introducing a return statement.

When a return statement is introduced the execution of a expression list may be interrupted if one of the expressions is a return statement. The return is to go all the way back to the last message send, resulting in the value of the return-statement to be returned to the message sender as the value of the executed method.

To model this three new functions/macros are defined for controlling the sequence of actions in the π-program:

P ifContinue Q == (stop continue)(P | stop(x) + continue(x).Q) where x is not in fn(Q)

Stop == $\underline{stop}$ stop.0

Cont == $\underline{continue}$ continue.0

All "old" expressions end with a Cont-call as for instance:

[[X]] (w,v) == $r_x(u).\underline{v}u$.Cont

[[self]] (w,v) == $\underline{v}w$.Cont

[[E M]] (w,v) == (v_1) ([[E]](w, v_1) | (u) ($v_1(w_2).\underline{w_2}u.\underline{u}m.u(z).\underline{v}z$.Cont))

The only expression using the Stop-macro is the new return expression:

[[↑ E]] (w,v) == (v') ([[E]] (w,v' | v'(z). $\underline{v}z$.Stop)

ifContinue is used in the definition of expression lists:

[[E1. ... En]](w,v) == (v')([[E1]](w,v') ifContinue ... ifContinue v'(e).[[En]](w,v') | v'(e).$\underline{v}e$)

7 Introducing control structures.

7.1 Control structures in Smalltalk

In Smalltalk control structures are made by defining methods for boolean objects which take objects holding a block of code as parameters. The central point in doing this is having such block-of-code-objects.

A typical control-expression is then:

```
bool ifTrue: [expressions].
```

where `bool` is a ref-variable to a boolean object, `ifTrue:` is a message to this object and `[expressions]` is a block object. The block object is created by the [..] - expression and may be send the message `value` to get the expressions executed and the value of the last expression returned. The method of `ifTrue`: in a class for true-objects may then look like:

```
ifTrue: aBlock == aBlock value.
```

and for the class for false-objects:

```
ifTrue: aBlock == <empty>
```

A while-constuct can be made making a method for block-objects:

```
whileTrue: aBlock ==
      b:= self value.b ifTrue: aBlock.self whileTrue: aBlock.
```

This may be used as follows:

```
[expression returning bool object] whileTrue: [expressions].
```

Blocks can also take parameters:

```
[:x | expressions]
```

and to execute this the message `valueWith: parameterExpression` is sent the block-object, which makes `expressions` execute with the parameter-value assigned to x. Blocks are also used when making methods for going through collections of objects doing some operations on each of them. Typically:

```
collection do: [:x | expressions]
```

where collection is an object holding other objects. The implementation of do:

```
do: aBlock▫==
      <for each element in self:>
      aBlock valueWith: element
```

7.2 Block objects

In the system there is a special class for such block-of-code-objects, it is called CompiledBlock. This class has a method which when called executes the code. The code may take parameters and always returns a reference-value. The block of code also belong in a context - that is the object and the method where it was created - so that it can reference variables outside the block. This is also represented by an object of class BlockContext. "The whole thing" (both the block and the context) is represented by an object of class BlockClosure.

Let us see how such objects may be expressed in the π-calculus:

BNF: ObjectCreation = ... | [ExpressionList] | [Ident* | ExpressionList]

There are three parameters to expression-functions to handle return expressions in blocks. These are: w is owner object, u is channel to return value for method (return for return-expressions), v is return channel for non-return expression (return for last expression in the block if it is not a return-statement).

[[↑ E]] (w,v,u) == (v') ([[E]] (w,v') | v'(z).$\underline{u}$z.Stop)

Other expressions are as before as the parameter u is not used. The translation of expression lists is changed:

[[E1. ... En]](w,v,u) == (v')([[E1]](w,v',u) ifContinue ... ifContinue v'(e).[[En]](w,v,u))

Block objects are created by getting a new object and then creating a block-process which can be run several times by sending a `value` message to the class.

[[[ExpressionList]]] (w,v) == (x) (c_{block}(n).$\underline{n}$x.$\underline{v}$n | (z) ($\underline{x}$z * z(u). [[ExpressionList]](w,v,u))).

Here first a block-obejct is created by c_{block}(n). This object is then sent a local channel to this process (= $\underline{n}$x), and then the channel to the block-object is returned as the value of the expression (= $\underline{v}$n). After that the process waits for someone to contact it (= $\underline{x}$z) and gets the local return channel (= z(u)) before the expression list is executed. Block with parameters are similar except they get the parameters before they start executing the expressions in the list:

[[[X_1...X_n | ExprList]]] (w,v) ==
(x) (c_{block}(n).$\underline{n}$x.$\underline{v}$n
| (z) ($\underline{x}$z * Decl(X_1...X_n)
| (x_1...x_n) (z(u).z(x_1).$\underline{w}_{x_1}x_1$...z(x_n)$\underline{w}_{x_n}x_n$. [[ExprList]](w,v,u))))

Block class == (w) ($\underline{c}_{block}$(w).w(x).
(u) ($\underline{w}$u * u: [
#value => (z) (x(z).$\underline{z}$u)
#valueWith: => (z)(p) (x(z).u(p).$\underline{z}$u.$\underline{z}$p)

#evalueWith:with:... =>(z) (p_1)...(p_n) (x(z).$\underline{z}$u.u(p_1).$\underline{z}$(p_1)...u(p_n).$\underline{z}p_n$])))

When translating block objects to π-calculus processes no special concern has to be taken for storing a blocks context as the block-process is located in a context "analogous" to the context created in Smalltalk. In this context object-variables and method-variables are available to the block-process / object. One discussion might be if the variables for the block-object should be in the same state when the block executes as when it was created (that is the variables must be copied and stored for the block) or if the variables may be changed in the time span between the blocks creation and its execution. The approach described here takes the latter stand point. If taking the first stand point some mechanism to copy and store the variables must be introduced.

8 Related work on object-oriented language semantics.

There have been two papers giving the semantics of Smalltalk: [Wol] and [Kam]. [Kam] focus is on describing inheritance denotationally and has a more "pure" denotational description than [Wol]. [Kam] use fixed point to define inheritance while [Wol] describes inheritance more operationally by similar notions to method-lookup in a class hierarchy. In this paper inheritance is defined by having one object potentially be a structure of objects, one for each sub/superclass. This is similar to inheritance description in [Sci], but in [Sci] this object-structure can be manipulated during runtime, something which is not the case here.

[Wol] has a simpler description than [Kam] as it does not include a return-statement. The complexities arise from the necessity of having to introduce continuations in order to handle returns.

[Wol] handles "messageNotUndersood" explicitly and initialise variables with a reference to 'nil' as is also done here, while [Kam] does not consider any error event and uses as default-value for pointers which is an "undefined" value.

As mentioned earlier this paper is influenced and tries to follow the style of [Walk]. [Walk] defines two languages L1 and L2. L2 is the language closest to Chat but there are several differences which are described below.

In L1 and L2 classes have code which is executed in an object when it is created. Chat (and Smalltalk) objects do not have such code. The only basic behaviour an object has is answering messages, corresponding to a contiguous execution of the L2 statement "answer(M_1...M_n)".

L1 and L2 have control-structures something which does not exist in Smalltalk. Instead execution sequence control is done by sending messages to boolean objects and also to other objects. An important feature needed to do this is the ability to create objects which can hold blocks of code. The objects execute the code they hold when sent a special message (in Smalltalk called ´value´). Such objects play an important part as parameters to execution sequence control messages which use the objects almost as functions, and pointers to these objects can be seen as pointers to functions. Having such code-holding objects in the language facilitates the creation of innovative and practical control-structures by implementing methods in appropriate classes. Such code holding objects are in Smalltalk called block objects. Treatment of block objects are described above, but is not a feature existing in L1 or L2.

In L1 and L2 a program is started by making an instance of the first class in the declaration. In Chat a program has a body which is what starts executing when the program is run.

Neither L1 nor L2 have classes with inheritance, but this is included in Chat.

[Walk] do not consider errors which can occur when sending a message to an object for which the receiver have no corresponding method. This is handled above in the Smalltalk-way by sending the receiver the message ´messageNotUnderstood´.

[Nier1] and [Nier2]] define object behaviour using Abacus [Nier3] which is based on Milner's CCS [Mil1] and Hoare's CSP [Hoare]. Systems with dynamic intercommunication structure or dynamic linkage are not directly representable in these languages so they only describe static aspects. This limitation is something which has also been identified by the CCS "makers" and the π-calculus is an "answer" to this (among other things). Since most object-oriented languages features dynamic structures and linkage it is easier to express the semantics of such a language in π-calculus than in something based on CCS and CSP.

9 Conclusions

The presented work shows that it is rather simple to express the denotation of an object-oriented language in a calculus of mobile processes. As seen in [Kam] it is not similarly simple to express it using denotational semantics. On this background one may perhaps conclude that objects are more closely related to the process-paradigm than to that of denotational semantics. It seems that languages of traditional non-ref variables, functions, procedures and control-structures fits with denotational semantics while processes and objects fits with process-calculus. Hybrid languages have half of each and there fore get more complex descriptions as seen if comparing L1/L2 descriptions to Chats - not only because there are more concepts to describe, but also because in describing the semantics one must choose one of the two as the basis and then have to use more cumbersome expressions to describe the concepts which are not in line with the basis.

Limiting the language to only have ref-type variables has simplified the semantic definition in π-calculus. Making this limitation on variable types may seem unpractical but the usability of languages like Smalltalk-80 shows that this is not always the case. Doing this limitation in a language has some advantages in that there are fewer concepts to learn and handle as a programmer and the runtime structures become more homogeneous which may makes it possible to make simpler and better programming tools such as editors and debuggers.

The work presented here also shows that using π-calculus instead of traditional denotation semantics simplifies the description of expressions like the return-statement which change the execution sequence. Continuations, which must be used for dentational semantics, are avoided and the description then becomes more readable.

In comparing the languages of Smalltalk and π-calculus one can notices that:

- Smalltalk has only one sting of execution while π have parallel events
- In Smalltalk the notion of objects identity is central, while a process in π (and similar languages) are just sequences of events sending and receiving "messages". The object identity may be seen as a channel the process/objects is always listening to, while a process may listen to varying channels over time. The channel named w in the descriptions presents the objects identity and the channel another process always can reach it through.
- The basic object oriented notion of a system consisting of objects which send messages is not directly visible in π because of the single concept of channel name being used for the basically different object oriented concepts: object identity, message name and variables. There fore π does not support object oriented concepts directly, however the above work shows that object oriented concepts are easily mapped to π concepts and some object oriented concepts are mapped to the same π concept.

Future work:

In the above description classes and methods are not objects, something which they are in Smalltalk. It would therefore be interesting to see how these aspects of Smalltalk van be expressed in the π-calculus.

By expressing the language semantics using the π-calculus addition of parallel features into the language can easily be expressed and studied. Such features may include:

- parallel execution of a block for different parameters,
- parallel sending of messages making the sender wait until all has returned,
- sending messages with no wait for return

(Adding expression for these thing into Smalltalk turns quasi-parallel objects into entities which may execute in real parallel, perhaps on parallel hardware and there by exploiting the hardwares computing powers something which is hard to do when having to make a program design based on a sequential programming language paradigm.)

In [Nier2] an object type theory is presented based on a labelled transition system. For future work a similar approach to typing of objects could be studied based on the definition of bisimulation in [Mil2/3].

Future work may also be to prove by bisimulation that delegation and inheritance can express the same things, and also that inheritance is similar to copying code directly down into each class, something which has an analogy in the fixed point operation for inheritance used in [Kam].

Acknowledgements

I thank Bjorn Kristoffersen for pointing out the π-calculus to me and Olaf Owe for valuable comments on an earlier version of this paper.

11 References

[Gold] Adele Goldberg, David Robson. Smalltalk-80: The language and its implementation, Readings, Mass: Addison-Wesley, 1983

[Kam] Samuel Kamin, W.Springfield, Inheitance in Smalltalk-80: A denotational Definition, ACM Symposium On Principles of Programming Languages 1988

[Hoare] C.A.R. Hoare, Communicating Sequencial Processes, Prentice Hall, 1985

[Mil1] Robin Milner, Communication and Concurrency, Prentice-Hall 1989

[Mil2] Robin Milner, Joachim Parrow, David Walker, A Calculus of Mobile Processes, Part I, Report ECS-LFCS-89-85, Laboratory of Foundations of Computer Science, Computer Science Department, Edinburgh University 1989

[Mil3] Robin Milner, Joachim Parrow, David Walker, A Calculus of Mobile Processes, Part II, Report ECS-LFCS-89-86, Laboratory of Foundations of Computer Science, Computer Science Department, Edinburgh University 1989

[Nier1] Oscar Nierstrasz, Michael Papathomas, Viewing Objects as Patterns of Communicating Agents, ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90

[Nier2] Oscar Nierstrasz, Michael Papathomas,Toward a Type Theory for Active Objects, in Object Management, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990

[Sci] Edward Sciore, Object Specialisation, ACM Transactions on Information Systems, Vol. 7, No.2, April 1989

[Wol] Mario Wolczko, Semantics of Smalltalk-80, Proceeding of ECOOP 1987

[Walk] David Walker, π-Calculus Semantics of Object-Oriented Programming Languages, Lecture Notes in Computer Science number 526.

Appendix Z Notes and comments (private)

-A set of agent identifiers is also presupposed, each with an arity - an integer ≥ 0. Below A range over agent identifiers.----------

COMMENT (Question) ???????
In the paper the m-channel names are restricted to the class where they are defined. My question is then how external processes representing other objects can send these messages ? It goes as follows:
- an external object sends a local channel name to a receiver object
- over this channel comes msg-name and parameters
- object receives msg and params
- then the receiving object "grabs" an out signal from the appropriate method
  and receives a local channel ($m_i(z)$ in Answer() see below)
- over this channel the "official" channel to the receiving object and parameters are sent
  to the method-process (z(w).z(p)..... see Answer() below)
- and when finished the method process uses this local channel for the return value
- the object receives the value and send this on the first local channel to the msg-sender object

Perhaps the question is: can channels be used as parameters even if they are restricted to other parts of the system ?

w(x) x(a) | (a) (a(v) *M | w(x).x(a). a(y)) --- does M get executed ? is this legal π-code ?

x(a) | (a) (a(v) *M | x(a). a(y)) (simplified version)

w is the official channel to the receiving object, a corresponds to msg/method name, x(a) is sending the message, x(a). a(y) is method lookup/call and a(w) *M correspond to the selected method.

version with two methods:

w(x) x(a) | (a,b) (a(v) * M1 | b(v) * M2 | w(x).x : [a⇒ a(y), b ⇒ a(y))

If the method/message names must be global in order for other objects to send them all methods must have unique m-channel names or else one would get randomly one of several methods having the same name (sending m(z) all of them).

Methods must have variable names distinct from instance variables - or does a restriction ($x_1...x_n$) hide any external overlapping x's ?