Syntax and Semantics of Pi in the Sky Calculus

John Glauert, Zurab Khasidashvili, Lone Leth, and Bent Thomsen

Version L. 1st December 1995

1 Changes

Version L integrates some thoughts on replication from a separate document by moving away from a π–calculus with agent definitions to one with a restricted form of replication. There is discussion of the varieties of π–calculus in use in the literature.

There is a significant change from unconditional input with a separate conditional construct to the use of conditional input. The previous forms are still available, but seen as syntactic sugar. Certain processes are made very much simpler as a result.

The history of earlier versions is in the final section.

2 Introduction

Awaits material giving proper motivation and a gentle introduction to the world of processes and messages we are considering.

The paper gives a definition of the syntax and semantics of π^π–calculus. We show that the notation is powerful enough to simulate the π–calculus, giving an encoding of π–calculus terms in the π^π–calculus. We also (will) prove that the encoding is correct. This encoding represents the names of the π–calculus, which can be though of as unbuffered communication channels, as π^π–calculus processes which act as mailboxes.

We (will) go on to show that any π^π–calculus system may be simulated using the π–calculus.

It has been demonstrated that the λ–calculus can be modelled using the π–calculus. Clearly, the π^π–calculus can model the λ–calculus through simulation of the corresponding π–calculus. However, a more direct translation is also presented and (will be) shown to be correct.

We (will) also give a translation of the π^π–calculus into Object Graph Rewriting (OGRe). OGRe is described in "Parallel Implementation through Object Graph Rewriting" by John Glauert.

The λ–calculus can be modelled in OGRe via the translation of the equivalent π^π–calculus term. We (will) also give a more direct translation.

3 π^π–Calculus Syntax

We assume a set N of *names* ranged over by n, m, etc. A set Σ of *symbols* ranged over by F, G, etc. each with an associated *arity* (≥ 0) giving the number of arguments it should take. These are combined in the set of *terms*, T ranged over by t:

$$T \quad ::= \quad n \mid F(t_1, \ldots, t_n)\ , \ \ where\, arity(F) = n$$

P, the class of process, and S, the class of process *scripts*, are defined in terms of each other.

3.1 π^π–Calculus Processes

Parallel *Processes* form the class P defined by the following syntax:

$$P \quad ::= \quad n : S \mid n!t \mid \nu n\, P \mid P \parallel P' \mid ()$$

$n : S$ is a named process (the idea is similar to locations in some versions of CCS) running a script S. $n!t$ is a message t on its way to process n. These two forms make up a π^π–calculus network and all computation involves the interaction of named processes and messages.

$\nu n\, P$ denotes restriction of the name n to P. P may contain a process with name n, but the name n is bound by the restriction and is not confused with any other use of the same name in the surrounding context.

$P \parallel P'$ denotes parallel composition of processes, and () is the empty composition. It is assumed that P and P' in $n!t$ are linear wrt $n : S$ subcomponents. This just means that named processes are unique. Rules to ensure this are given below.

3.2 π^π–Calculus Scripts

Process *Scripts*, S are given by the following syntax:

$$S \quad ::= \quad \sum_i ?t_i \cdot S_i \mid \lambda n\, P \mid A(t_1, \ldots, t_n)$$

Also a set of recursive *agent* definitions is assumed: $A(n_1, \ldots, n_k) = S$.

The intuition is that S is the sequential body of a process. An input–guarded script takes the form $\sum_i ?t_i \cdot S_i$. If a message is directed to the process and matches at least one of the patterns t_i then the process continues with the script given by the corresponding S_i. Variables in t_i bind over S_i. If more than one pattern matches, the first is taken.

Abstractions of the form $\lambda n\, P$ allow a process to generate new processes. The parent process continues to exist if there is a process named n within P. That process takes on the name of the parent and other processes in P are spawned in parallel. We will illustrate the power of the construct with some examples:

To spawn a message t directed to process n and continue with a script S we can use an abstraction $\lambda m\, (n!t \parallel m : S)$.

To spawn a new process running script S', and continue with script S we can use an abstraction $\lambda m\, \nu n\, (n : S' \parallel m : S)$.

More complex process networks involving numbers of messages may be created in a similar way. Abstractions also allow a process to discover its identity. The script S in the following abstraction finds n bound to its process name: $\lambda n\, (n : S)$.

To terminate a process we can invoke a definition *Stop* where $Stop = \lambda n\, ()$. An agent *Stop* with this definition is assumed to exist in the rest of the paper.

Recursive scripts are dealt with through agent definitions. { I am tempted to say that all free names in the body of the definition should appear in the head, but as long as capture of free names is prevented during substitution, I think it would be possible for free names to appear. These free names would identify processes existing in the original process configuration. }

In previous versions we permitted the match construct in scripts:

$$[n = t \ : \ S]; S'$$

which tests to see if the term bound to n matches the term t. The term contains one or more symbols, and may introduce variables. The construct behaves as S if the term bound to n matches t. Names in n are bound in S by a successful match. If the match fails it behaves as S'.

This is seen as syntactic sugar in the present version. For each such match we assume a new symbol $Patt$. Then

$$[n = t \ : \ S]; S' \quad \equiv \quad \lambda m \, (m!Patt(n) \parallel m : (?Patt(t) \cdot S + ?Patt(x) \cdot S'))$$

where m, and x do not occur in S or S'. This script sends to itself a message which will match the first case if n is an instance of t and the second case otherwise. No other message sent to this process will use the symbol $Patt$.

3.3 π^π–Calculus Syntactic Constraints

It is necessary to define concepts of free and bound names in π^π–calculus. We will also describe some static constraints to be obeyed by well–formed processes and scripts.

Free names of processes are given by $fn(P)$. $dn(P)$ identifies defining names – free names which name processes in P.

Free names of scripts are given by $fn(S)$. Names in a term, t, are denoted by $fn(t)$. No free names in scripts or terms are defining names.

$$\begin{aligned}
fn(n : S) &= fn(S) \cup \{n\} \\
dn(n : S) &= \{n\} \\
fn(n!t) &= fn(t) \cup \{n\} \\
dn(n!t) &= \{\} \\
fn(\nu n \, P) &= fn(P) \setminus \{n\} \\
dn(\nu n \, P) &= dn(P) \setminus \{n\} \\
fn(P \parallel P') &= fn(P) \cup fn(P') \\
dn(P \parallel P') &= dn(P) \cup dn(P') \\
fn(()) &= \{\} \\
dn(()) &= \{\}
\end{aligned}$$

$$\begin{aligned}
fn(\sum_i ?t_i \cdot Si) &= \bigcup_i fn(S_i) \setminus fn(t_i) \\
fn(\lambda n \, P) &= fn(P) \setminus \{n\} \\
fn(A(t_1, \ldots, t_k)) &= fn(t_1) \cup \ldots \cup fn(t_k)
\end{aligned}$$

To ensure that named processes have distinct names, we need to add the constraint that $dn(P) \cap dn(P') = \{\}$ in $P \parallel P'$. We also want to make sure that every new process created by use of an abstraction in a script is bound by some restriction. In $\lambda n \, P$, we require that $dn(P) \subseteq \{n\}$. Hence the only possible free named process is n which is bound in the abstraction.

For parametric definitions $A(n_1, \ldots, n_k) = S$ we require that $fn(S) \subseteq \{n_1, \ldots, n_k\}$.

Note that the consequence is that no free name in a script is a defining name. Hence, such a name cannot be bound by input or matching.

One rather operational choice would be to ensure that every potential message has a process to go to. This would mean that a process could not just die by using a script like $Stop$. We would need a new form of script for a terminated process, say $Dead$, and a structural equivalence rule to garbage collect processes when the form $\nu n\,(n : Dead)$ arises.

We would need extra constraints to say that whenever there is a restriction $\nu n\,P$, $n \in dn(P)$. The constraint for an abstraction, $\lambda n\,P$, would be strengthened to read $dn(P) = \{n\}$.

4 π^π–Calculus Semantics

A sort discipline seems necessary to ensure that if an agent definition contains $n!t'$ then n will be bound to a defined name when the definition is expanded.

The following structural rules define structural congruence of π^π–calculus processes:

$$\begin{array}{rcll}
\nu n'\,\nu n\,P & \equiv & \nu n\,\nu n'\,P & \\
\nu n\,P & \equiv & \nu m\,P\{m/n\}\,, & m \notin fn(P) \\
\nu n\,P \parallel P' & \equiv & \nu n\,(P \parallel P')\,, & n \notin fn(P') \\
\nu n\,() & \equiv & () & \\
(P \parallel P') \parallel P'' & \equiv & P \parallel (P' \parallel P'') & \\
P \parallel P' & \equiv & P' \parallel P & \\
P \parallel () & \equiv & P & \\
n : A(t_1, \ldots, t_k) & \equiv & n : S\{t_1/n_1, \ldots, t_k/n_k\}\,, & where\ A(n_1, \ldots, n_k) = S \\
n : \lambda m\,P & \equiv & P\{n/m\}\,, & dn(P) \subseteq \{m\}
\end{array}$$

The final rule applies an abstraction and may involve process spawning and message creation.

The reduction semantics is given by a single reduction rule on processes P:

$$n!t \parallel n : \sum_i ?t_i \cdot S_i \quad \rightarrow \quad n : \sigma_j(S_j)\,, \qquad \exists j \leq i, \sigma_j \,:\, t = \sigma_j(t_j) \wedge \forall k < j, \sigma \,:\, t \neq \sigma(t_k)$$

σ is a mapping from names to terms. A message is received by process n if it has a script ready for input with a pattern matching the message. Since patterns might overlap, the earliest pattern will be chosen.

The following rules extend the reduction relation to contexts:

$$\begin{array}{rcl}
P \rightarrow P' & \Rightarrow & \nu n\,P \rightarrow \nu n\,P' \\
P \rightarrow P' & \Rightarrow & P \parallel Q \rightarrow P' \parallel Q \\
Q \equiv P,\ P \rightarrow P',\ P' \equiv Q' & \Rightarrow & Q \rightarrow Q'
\end{array}$$

We have not discussed the initial configuration of a process network in this calculus. If we disallow free names in agent definitions, then, without loss of generality, it could take the form $\nu n\,(n : A)$ for some agent A of arity 0.

5 Review of π–calculus

The π–calculus appears in various syntactic forms. The presentation of semantics has also evolved.

The monadic form in [MPW89] provides very general choice, a matching construct, and parametric definitions. The simpler, more restricted, form in [Mil90] uses replication and omits choice and matching. It is sufficiently powerful to model the λ–calculus, however.

The Polyadic π–calculus in [Mil91] has guarded choice and replication. Milner shows that conditional computation may be modelled without a match operation and that parametric definitions can be (weakly) simulated by replication. Polyadic communication can be seen as syntactic sugar based on monadic communication.

Meanwhile Honda an Tokoro [HoTo91] and Boudol [Boudol] (and Glauert [Gla92]) show that the asynchronous calculus, in which processes are not guarded by output actions, has all the required power, and may be used to simulate the synchronous calculus in almost all respects.

There is increasing use of a simplified calculus which is called the *Mini Asynchronous π–Calculus*. This has no choice or match construct. Communication is asynchronous and a restricted form of input-guarded replication is used.

We will use such a syntax for the π–calculus, though we will consider polyadic communication and synchronous communication. Both of these extensions can be seen as syntactic sugar, but can also be simulated directly in the π^π–calculus.

$$N \quad ::= \quad x(\vec{y}) \cdot N \mid !\, x(\vec{y}) \cdot N \mid \overline{x}\,\vec{v} \cdot N \mid \overline{x}\,\vec{v} \mid (\nu\, x)\,(N) \mid N \mid N' \mid 0$$

All features of the full π–calculus can be simulated by the mini asynchronous π–calculus, except perhaps the general choice construct which does not seem to be of interest to anyone. For example, we can provide something like the general case of replication. In the following, $x \notin fn(P)$:

$$Q = (\nu\, x)\,(\overline{x} \mid !\, x() \cdot (P \mid \overline{x})) \quad \approx \quad !\, P$$

Q becomes, by a τ step,

$$\begin{array}{rl} & (\nu\, x)\,(P \mid \overline{x} \mid !\, x() \cdot (P \mid \overline{x})) \\ \cong & P \mid (\nu\, x)\,(\overline{x} \mid !\, x() \cdot (P \mid \overline{x})) \\ = & P \mid Q \end{array}$$

So it seems that Q is weakly bisimilar to $!\, P$ using standard π–calculus semantics.

6 Simulation of π–calculus by π^π–calculus

A proposed translation of the π–calculus is provided. The translation is based on the use of a π^π–calculus process to implement a π–calculus name (communication channel). A π^π–calculus channel process coordinates requests for input and output actions using the name. Input requests are provided by *Get* messages which carry the identity of the process to receive data. Output requests are provided by *APut* messages which carry data values.

$$Chan \quad = \quad ?Get(n) \cdot ?APut(v) \cdot \lambda l\,(n!v \parallel l : Chan)$$

Originally, a *Chan* process will only take a *Get* message. Outstanding *APut* messages are only accepted once a *Get* has been seen. A data message is sent and the script is repeated.

This form supports asynchronous communication. Synchronous communication requires that the sending process is blocked until the communication completes. This is achieved by making the translation of the output guarded process wait for a synchronisation signal $Sync$ when the communication is closed. We use Put messages in place of $APut$ and add the identity of a process to receive the synchronisation message.

$$Chan \;=\; ?Get(n) \cdot ?Put(v,s) \cdot \lambda l\,(n!v \parallel s!Sync \parallel l : Chan)$$

In the translation below, we support both styles, at the cost of a slightly more sophisticated definition of $Chan$:

$$Chan \;=\; ?Get(n) \cdot (?APut(v) \cdot \lambda l\,(n!v \parallel l : Chan) + ?Put(v,s) \cdot \lambda l\,(n!v \parallel s!Sync \parallel l : Chan))$$

In the following we only handle monadic communication. Jeong Ho has extensions to handle the polyadic form. The function t maps π–calculus processes to π^π–calculus processes:

$$\begin{array}{rcl}
t[\![x(y) \cdot N]\!] & = & \nu m\,(x!Get(m) \parallel m : ?y \cdot \lambda m'\, t[\![N]\!]) \\
t[\![!\,x(y) \cdot N]\!] & = & \nu m\,(m : A(x, \vec{\mathrm{N}})) \\
\textbf{where} & & A(x, \vec{\mathrm{N}}) \;=\; \lambda m\,(x!Get(m) \parallel m : ?y \cdot \lambda m'\,(m' : A(x, \vec{\mathrm{N}}) \parallel t[\![N]\!])) \\
 & & \vec{\mathrm{N}} \stackrel{\triangle}{=} (fn(N) \setminus \{x, y\}) \\
t[\![\overline{x}\, v \cdot N]\!] & = & \nu m\,(x!Put(v,m) \parallel m : ?Sync \cdot \lambda m'\, t[\![N]\!]) \\
t[\![\overline{x}\, v]\!] & = & x!APut(v) \\
t[\![(\nu\, x)\,(N)]\!] & = & \nu x\,(x : Chan \parallel t[\![N]\!]) \\
t[\![N \mid N']\!] & = & t[\![N]\!] \parallel t[\![N']\!] \\
t[\![0]\!] & = & ()
\end{array}$$

The names in the π–calculus are represented by π^π–calculus processes which start by running the script $Chan$. It is the restriction construct which causes the channel processes to be introduced. If a π–calculus term with a free name is translated, no corresponding channel process is produced. Hence we need to consider carefully how a π–calculus term is simulated by the π^π–calculus.

I suggest a possible framework. How do we investigate the properties of a π–calculus term, P? In general, we might perform an experiment by placing P in parallel with some probe, represented by Q, and observe the behaviour of the combined system. Without loss of generality, we can restrict free names of the combination:

$$(\nu\, \vec{a})\,(P \mid Q) \quad where \quad \vec{a} = fn(P \mid Q)$$

Hence we would be considering the translation of closed π–calculus terms. If such terms are encoded, suitable π^π–calculus channel processes will always be generated.

This may be adequate if we are only interested in whether systems converge or not. However, if we consider a labelled transition system for π^π–calculus we will only observe τ events from such systems.

A different approach is simply to add π^π–calculus processes of the form $n : Chan$ for each name $n \in fn(P)$ in parallel with the translation of π–calculus term P.

7 A Restricted Subcalculus of π–Calculus

In some very restricted circumstances we do not need to use the translation based on special π^π–calculus channel processes. In these circumstances we can be sure that for each name, at most one π–calculus process in a parallel composition of processes will have an input prefix for that name at a given time. It will be seen that this imposes severe restrictions on the possible forms of π–calculus process, but these restrictions are obeyed by networks generated by some simulations of the λ–calculus.

A first observation is that a process cannot receive input on any name it has received as input – unless we are to trace the behaviour of all processes which might send output on a given name. Secondly, in a parallel composition only one process in the composition can take input on a given name.

Every symbol, A, used in parametric definitions has an arity, where $0 \leq arity(A)$. In the restricted calculus we will also associate an input arity, $0 \leq inarity(A) \leq arity(A)$, which identifies the parameters which can carry names used for input.

We will define a function $dn(N)$ over π–calculus processes. This will be similar in effect to the function defined for π^π–calculus and will be used to constrain the form of π–calculus processes we allow:

$$\begin{aligned}
dn(x(y) \cdot N) &= \{x\} \\
dn(!\,x(y) \cdot N) &= \{x\} \\
dn((\nu\, x)\,(N)) &= dn(N) \setminus \{x\} \\
dn(\overline{x}\, v \cdot N) &= dn(N) \\
dn(\overline{x}\, v) &= \{\} \\
dn(N \mid N') &= dn(N) \cup dn(N') \\
dn(0) &= \{\}
\end{aligned}$$

We then add some constraints that must be obeyed. Firstly, in $(N \mid N')$, we require that $dn(N) \cap dn(N') = \{\}$. Hence no free name may be used for input in both N and N'.

Secondly, in $n(n') \cdot N$, we require that $dn(N) \subseteq \{n\}$. Otherwise, after performing input, the property required for a parallel composition might be violated. A consequence is that if N contains a parallel composition, all names used for input must be restricted, except for n.

For similar reasons, in $!\,x(y) \cdot N$ we require that $dn(N) = \{\}$. Unravelling a replication only happens following an input action, in which case we have $(N\{v/y\} \mid !\,x(y) \cdot N)$, where v is the value received. In the new network the replication can receive on x so we require that $N\{v/y\}$ cannot.

8 Simulation of λ–calculus by π^π–calculus

It has been demonstrated that the λ–calculus can be modelled using the π–calculus. Clearly, the π^π–calculus can model the λ–calculus through simulation of the corresponding π–calculus. However, a more direct translation is also presented and (will be) shown to be correct.

9 Simulation of π–calculus by OGRe

We give a translation of the π–calculus into Object Graph Rewriting (OGRe). The translation below is based on the use of an OGRe process to implement a π–calculus name (communication channel).

9.1 OGRe Channel Processes

OGRe may be used to implement a channel process which receives either requests for input, provided by *Get* messages, or data values, with optional synchronisation addresses, provided by *APut* and *Put* messages. *Get* messages contain the names of OGRe processes to receive data values. *APut* messages contain a value to be sent. In addition to a value to be sent, *Put* messages also contain the name of an OGRe process to receive a synchronisation message when the communication completes.

Initially, a π–calculus channel n will be represented by a process with state $Chan$:

$$n : Chan$$

$$\begin{array}{rcl} c : Chan,\ c\,!\,Get(r) & \rightarrow & c : RChan(r) \\ c : RChan(r),\ c\,!\,APut(v) & \rightarrow & c : Chan,\ r\,!\,Data(v) \\ c : RChan(r),\ c\,!\,Put(v,s) & \rightarrow & c : Chan,\ r\,!\,Data(v),\ s\,!\,Sync \end{array}$$

In the translation of a π–calculus term, an input action is simulated by a *Get* message which carries the identity of an OGRe process which will simulate the term following the action. This process will wait for a *Data* message carrying the value from the corresponding output action when a communication is closed.

An output-guarded action is simulated by a *Put* message to the OGRe channel process corresponding to the subject of the action. The *Put* message carries the value offered and the identity of an OGRe process which will simulate the π–calculus term following the action. This process will wait for a *Sync* message indicating that the communication has been closed and the value passed to the corresponding input action.

Asynchronous output is simulated using an *APut* message which requires no synchronisation.

9.2 Simulation of π–calculus

The function q maps π–calculus processes to OGRe processes. The parameter m is the name of the OGRe process corresponding to the translated π–calculus process:

$$\begin{array}{rcl} q[\![x(y)\cdot N]\!]_m & = & A(\vec{\mathrm{N}}),\ x\,!\,Get(m) \\ & & \textbf{where}\quad m : A(\vec{\mathrm{N}}),\ m\,!\,Data(y) \rightarrow q[\![N]\!]_m \\ & & \vec{\mathrm{N}} \stackrel{\triangle}{=} (fn(N) \setminus \{y\}) \\ q[\![!\,x(y)\cdot N]\!]_m & = & A(x,\vec{\mathrm{N}}),\ x\,!\,Get(m) \\ & & \textbf{where}\quad m : A(x,\vec{\mathrm{N}}),\ m\,!\,Data(y) \rightarrow m : A(x,\vec{\mathrm{N}}),\ x\,!\,Get(m),\ m' : q[\![N]\!]_{m'} \\ & & \vec{\mathrm{N}} \stackrel{\triangle}{=} (fn(N) \setminus \{x,y\}) \\ q[\![\overline{x}\,v\cdot N]\!]_m & = & B(\vec{\mathrm{N}}),\ x\,!\,Put(v,m) \end{array}$$

$$
\begin{array}{rcl}
 & & \textbf{where} \quad m : B(\vec{\mathrm{N}}),\ m\,!\,Sync \rightarrow m : q[\![N]\!]_m \\
 & & \vec{\mathrm{N}} \stackrel{\triangle}{=} fn(N) \\
q[\![\overline{x}\,v]\!]_m & = & Dead,\ x\,!\,APut(v) \\
q[\![(\nu\,x)\,(N)]\!]_m & = & q[\![N]\!]_m,\ x : Chan \\
q[\![N \mid N']\!]_m & = & q[\![N]\!]_m,\ m' : q[\![N']\!]_{m'} \\
q[\![0]\!]_m & = & Dead
\end{array}
$$

The state $Dead$ labels a process which should never receive any further messages. It remains to be proved that this is the case for every $Dead$ process which arises from evaluating the OGRe system generated by q from any π–calculus term.

The names in the π–calculus are represented by OGRe processes of the form $Chan$. It is the restriction construct which causes the channel processes to be introduced. When a π–calculus term with a bound name is translated by the scheme q, a channel process is produced by the translation of the binding restriction.

To account for names which are free in the whole translated term, we add an OGRe process of the form $n : Chan$ for each name $n \in fn(P)$ in parallel with the translation of P. The translation of a π–calculus term P is $m : q[\![P]\!]_m$ where $m \notin fn(P)$.

The following example shows translation of the term $R = (x(y) \cdot P \mid \overline{x}\,v \cdot Q)$ where $fn(P) = \{y\}$ and $fn(Q) = \{\}$:

$$
\begin{array}{rcl}
m : q[\![x(y) \cdot P \mid \overline{x}\,v \cdot Q]\!]_m & = & m : q[\![x(y) \cdot P]\!]_m,\ m' : q[\![\overline{x}\,v \cdot Q]\!]_{m'} \\
 & = & m : A,\ x\,!\,Get(m),\ m' : q[\![\overline{x}\,v \cdot Q]\!]_{m'} \\
 & & \textbf{where} \quad m : A,\ m\,!\,Data(y) \rightarrow m : q[\![P]\!]_m \\
 & = & m : A,\ x\,!\,Get(m),\ m' : B,\ x\,!\,Put(v, m') \\
 & & \textbf{where} \quad m : A,\ m\,!\,Data(y) \rightarrow m : q[\![P]\!]_m \\
 & & \textbf{and} \quad\ \ m' : B,\ m'\,!\,Sync \rightarrow m' : q[\![Q]\!]_{m'}
\end{array}
$$

Since x is free in R we also add the OGRe process $x : Chan$. Assuming the definitions of A and B as defined above, the network may then evolve as follows:

$$
\begin{array}{rl}
 & m : A,\ x\,!\,Get(m),\ m' : B,\ x\,!\,Put(v, m'),\ x : Chan \\
\rightarrow & m : A,\ m' : B,\ x\,!\,Put(v, m'),\ x : RChan(m) \\
\rightarrow & m : A,\ m' : B,\ x : Chan,\ m\,!\,Data(v),\ m'\,!\,Sync \\
\rightarrow & m : q[\![P\{v/y\}]\!]_m,\ m' : B,\ x : Chan,\ m'\,!\,Sync \\
\rightarrow & m : q[\![P\{v/y\}]\!]_m,\ m' : q[\![Q]\!]_{m'},\ x : Chan
\end{array}
$$

Since, by the assumptions we made about free names, x does not appear free in P or Q, the channel process $x : Chan$ can never receive further messages and can be garbage collected. The resulting network is the same as $m : q[\![P\{v/y\} \mid Q]\!]_m$. $(P\{v/y\} \mid Q)$ is, of course, the result of a communication step in the original term $(x(y) \cdot P \mid \overline{x}\,v \cdot Q)$. Since π–calculus names are preserved in the translation it will be seen that we were justified in identifying $q[\![P]\!]_m\{v/y\}$ with $q[\![P\{v/y\}]\!]_m$.

10 Simulation of π^π–calculus by OGRe

We (will) also give a translation of the π^π–calculus into Object Graph Rewriting (OGRe).

11 Simulation of λ–calculus by OGRe

The λ–calculus can be modelled in OGRe via the translation of the equivalent π^π–calculus term. We (will) also give a more direct translation.

12 History

Version K introduced a translation of π–calculus into OGRe. Since we have a translation of π–calculus to π^π–calculus, the original aim was to provide a mapping to OGRe by developing a mapping from π^π–calculus to OGRe. This proved less straightforward than expected, while a direct mapping to OGRe presented few problems.

It remains to add a mapping from OGRe to π^π–calculus which should compose with the π–calculus to OGRe mapping to provide an alternative, but obviously equivalent, π–calculus to π^π–calculus mapping.

The document also omits a proper definition of OGRe which will be provided separately.

Version J added a little more explanation to Version I. It also introduced a running example showing how replication is modelled by a parametric definition and how such a definition can satisfy the constraints of the sub–calculus. Davide Sangiorgi seems to use parametric definitions in his thesis. We will restrict ourselves to the main use of replication which is always guarded by an input action.

Version I changed the reduction relation. If we are to use *barbed-bisimulation* related the reduction semantics, then the reduction relation should only capture the observable events. I think that means just communication, for now.

Version I introduced some definitions of free and bound names and also some static constraints to be obeyed by well-formed processes. I am not entirely happy with this and would value proposals for improvement! One major change is to try to live without a separate class of variables in terms. There is a bit more explanation of the process and script classes.

An extra section is added, defining a sub–calculus of the π–calculus which may be helpful in defining a more direct translation to π^π–calculus. If it could be shown that all π–calculus can be simulated in the sub–calculus, then an even simpler translation should be possible, since the sub–calculus is intended to capture the essential properties of π^π–calculus.

Version H contained minor corrections and a new formulation of the translation of the π–calculus which maps to π^π–calculus *processes* rather than *scripts*.

Version G works with the formulation of π^π–calculus introduced under the name term–π^π–calculus in *Version 6* which aims to have a more uniform syntax using a much simplified syntactic class for *scripts*. However, the central role of agent definitions is softened by the introduction of *abstractions* over process names.

References

[Boudol] G. Boudol

[Gla92] J.R.W. Glauert: *Asynchronous Mobile Processes and Graph Rewriting*, Proc PARLE'92. LNCS 605. June 1992. (1992)

[HoTo91] K. Honda & M. Tokoro: *An object calculus for asynchronous communication.* Proc. ECOOP'91, Geneva, July 1991.

[Mil90] R. Milner: *Functions as Processes*, Automata, Languages, and Programming. Springer LNCS 443. (1990) Also: Technical Report INRIA Sophia Antipolis, June 1989.

[Mil91] R. Milner: *The π^{π}–Calculus: A Tutorial*, Technical Report ECS-LFCS-91-180, Edinburgh University, October 1991. (1991)

[MPW89] R. Milner, J. Parrow, & D. Walker: *A Calculus of Mobile Processes*, Parts I and II, Technical Report ECS-LFCS-89-85, Edinburgh University, June 1989. (1989)