Pi-Calculus: A Unifying Framework for Programming Paradigms

Srinivas Bangalore
University of Pennsylvania

Pi-Calculus: A Unifying Framework for Programming Paradigms

Abstract

π-calculus is a calculus for modeling dynamically changing configurations of a network of communicating agents. This paper studies the suitability of π-calculus as a unifying framework to model the operational semantics of the three paradigms of programming: functional, logic and imperative paradigms. In doing so, the attempt is to demonstrate that π-calculus models a primitive that is pervasive in the three paradigms and to illustrate that the three forms of sequential computing are special instances of concurrent computing.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-81.

π-calculus: A Unifying Framework for Programming Paradigms

MS-CIS-93-81
LINC LAB 256

Srinivas Bangalore

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

September 1993

π-calculus : A Unifying Framework for Programming Paradigms *

Srinivas Bangalore
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

September 8, 1993

Abstract

π-calculus is a calculus for modeling dynamically changing configurations of a network of communicating agents. This paper studies the suitability of π-calculus as a unifying framework to model the operational semantics of the three paradigms of programming : functional, logic and imperative paradigms. In doing so, the attempt is to demonstrate that π-calculus models a primitive that is pervasive in the three paradigms and to illustrate that the three forms of sequential computing are special instances of concurrent computing.

*This work is submitted as partial fulfillment of the requirements of the WPEII. (Doctoral Written Preliminary Examination)

1 Introduction to π-calculus

A model of computation chooses to focus on certain phenomena that seem to be pervasive of computing and treats them to be the essence of computing. For example the λ-calculus models computation as a function that takes in arguments and yields results. A calculus for concurrent computation, on the other hand, treats computation as a communicating system in which communicating agents have the ability to interact and influence the behavior of one another. While such a model of computation cannot be forced into the function-argument paradigm of computation without loss of naturalness, a function-argument interaction behavior of agents would be a special case of interaction between agents in a model of concurrent computation.

π-calculus is a calculus for modeling dynamically changing configurations of a network of communicating agents. The key notion that underlies π-calculus' attempt to model concurrency is *naming* of entities. Naming provides an identity to an entity that allows it to concurrently coexist in an environment with other entities. π-calculus is unique in that it treats names of channels that the agents communicate on, as primitives instead of names of agents. Another distinctive feature of π-calculus is that it does not allow agents to be transmitted along communication channels, instead attempts to demonstrate that passing names of channels as contents of communication is in itself sufficiently general.

The objective of this paper is to study the suitability of π-calculus as a unifying framework to model the operational semantics of three differing paradigms of programming : functional, logic and imperative paradigms. In doing so, the attempt is to demonstrate that π-calculus models a primitive that is pervasive in the three paradigms and to illustrate that the three forms of sequential computing are special instances of concurrent computing. We shall provide an introduction to π-calculus based on [Milner 1991] and examine the usefulness of its primitives in modeling some of the programming concepts. Later we study the translation of representative languages of each of the three programming paradigms into π-calculus respecting their operational semantics as closely as possible.

1.1 Syntax of π-calculus

The primitive elements of π-calculus are structureless entities called *Names*, infinitely many and denoted by $x, y, z \ldots \in \mathcal{N}$. A name refers to a communication link or channel. The name x represents the input end of a channel x and the co-name $\overline{x}$ represents its output end. In the syntax that follows, agent identifiers $A, B, \ldots$ range over $\mathcal{K}$, and $P, Q, \ldots$ range over process expressions.

$$\begin{array}{lll} P & ::= & 0 \\ & | & \overline{x}y.P \\ & | & x(y).P \\ & | & \tau.P \\ & | & P + Q \\ & | & P \mid Q \\ & | & (\nu x)P \\ & | & [x = y]P \\ & | & A(y_1, \ldots, y_n) \end{array}$$

A *process* is an entity that derives its denotation from its interaction with other processes.

A process interacts with the other processes by receiving and transmitting names of channels. Processes can be characterized as follows based on their interaction behavior.

1. A process Q that performs no interaction is called *inaction* and is represented by 0.

2. $\overline{x}y.P$ represents a process Q that outputs the name y on channel x and behaves like P. The co-name x, may be regarded as the output port of the process Q. Since the name y is free in Q, the output action is called a *free output action.*

3. $x(y).P$ is a process Q that receives a name z, on its input port x and behaves like $P\{z/y\}$ (y for z in P). Such an action is called an *input action.* Though the action performed by Q appears to be similar to λ-application it differs from it in that y may be bound *only* to names and not to arbitrary terms.

4. $\tau.P$ represents a process Q that performs an internal action τ and then behaves like P. This action is called *silent action* since it does not involve other processes in the environment.

A summand process $P + Q$, represents a process that can behave as either P or Q, but it cannot commit to either of the alternatives until one of it occurs. This construct introduces one kind of non-determinism into process behavior.

The par ($|$) construct introduces a second kind of non-determinism into process behavior. The form $P \mid Q$ (read as P par Q), serves to model concurrent execution of two processes P and Q. P and Q may act independently but can also communicate with each other to perform a silent action. For example, if

$$P = \overline{x}y.P' \quad \textit{and} \quad Q = x(z).Q'$$

then the composition process $P \mid Q$ could either output on channel x, or could input from channel x. It can also interact internally (intra-act), performing a silent action with the residue process as $P' \mid Q'\{y/z\}$.

The form, $(\nu x)P$ (read as "new x") is a unique construct introduced by π-calculus. It restricts the name x to the process P thus making x private to P. Alternately, it declares a new name in P that is different from all external names. As a result the port x, that may appear in P becomes inaccessible to the environment. Thus, $(\nu x)x(y).0$ cannot communicate with any other process.

A match $[x = y]P$ is a process that behaves like P if x and y are identical otherwise like 0. For example,

$$P = \overline{x}y.P' \quad \textit{and} \quad Q = x(u).[u = w]Q'$$

then $(\nu x)(P \mid Q)$ communicates internally on channel x and results in $[y = w]Q'$. Since the match fails, this is identical to 0. Hence the final residue is $(\nu x)P'$.

Agents are parameterized processes. An agent, A, is uniquely defined by a defining equation of the form

$$A(x_1, \ldots, x_n) \overset{\text{def}}{=} P$$

where the names $x_1, \ldots, x_n$ are distinct and are the only names that may occur free in P. Then $A(y_1, \ldots, y_n)$ behaves like $P\{y_1/x_1, \ldots, y_n/x_n\}$ or $P\{\tilde{y} \;/\; \tilde{x}\}$. Substitution $\{\tilde{y} \;/\; \tilde{x}\}$, may require a change of bound variable names in P so as to avoid any of the y_i from becoming bound in P. Defining equations provide recursion, since P may contain any agent, even A itself. Also, we note that processes are zero-ary agents.

1.2 Rules for Actions

A process P, performs an action α and evolves into Q. This transition is denoted as

$$P \xrightarrow{\alpha} Q$$

Based on the prefix of P, we have seen that $\alpha \in \{\tau, \overline{x}y, x(y)\}$. A fourth action, called *bound output action* is possible when a process outputs a private variable on its output port. Such an action results in widening of scope of the private variable beyond the process it prefixes. For example, a bound output action results when a process such as $(\nu y)\overline{x}y.P$ communicates on channel x.

The silent action and free output actions are collectively termed *free actions* while input actions and bound output actions are called *bound actions.* The free and bound outputs are collectively called *output actions.* The x in $x(y).P$ or $\overline{x}y.P$ is called the *subject* and y is called the *object.*

Free names of P, $fv(P)$, are all the names that are not bound either by restriction or input prefix. The set of names of P, $v(P)$, is a union of its free and bound variables.

We define a transition relation to be the smallest relation that satisfies the rules of action given below.

$\tau - action$	:	$\dfrac{}{\tau.P \xrightarrow{\tau} P}$	
$output - action$	:	$\dfrac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$	
$input - action$	:	$\dfrac{}{x(w).P \xrightarrow{x(y)} P\{y/w\}}$	$y \notin fv((\nu w)P)$
$sum_{\mathcal{L}}$	:	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	
$sum_{\mathcal{R}}$	:	$\dfrac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$	
$match$	:	$\dfrac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$	
ide	:	$\dfrac{P\{\widetilde{y} / \widetilde{x}\} \xrightarrow{\alpha} P'}{A(\widetilde{y}) \xrightarrow{\alpha} P'}$	$A(\widetilde{x}) \overset{\mathbf{def}}{=} P$
$Par_{\mathcal{L}}$	:	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	$bv(\alpha) \cap fv(Q) = \phi$
$Par_{\mathcal{R}}$	:	$\dfrac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$	$bv(\alpha) \cap fv(P) = \phi$
Com	:	$\dfrac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}}$	
$Close$	:	$\dfrac{P \xrightarrow{\overline{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')}$	
Res	:	$\dfrac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'}$	$y \notin v(\alpha)$
$Open$	:	$\dfrac{P \xrightarrow{\overline{x}y} P'}{(y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}}$	$y \neq x$ *and* $w \notin fv((\nu y)P')$

1.3 Notation

We shall adopt the following notational convenience in the discussion to follow. We shall denote

- $\overline{x}y.0$ as $\overline{x}y$
- $x(y).0$ *as* $x(y)$
- $(\nu x_1)(\nu x_2)\ldots(\nu x_n)P$ *as* $(\nu x_1\, x_2 \ldots x_n)P$
- $(\nu y)\overline{x}y.P$ as $\overline{x}(y).P$
- communications that need to carry no parameters such as $x().P$ as $x.P$ and $\overline{x}\epsilon.P$ as $\overline{x}.P$.
- communications that need to carry more than one parameter by $x(y_1, \ldots, y_n)$ and $\overline{x}y_1, \ldots, y_n$
- multiple match patterns such as $x(y).([y = z_1]P_1 + \cdots + [y = z_n]P_n)$ as $x : [z_1 => P_1; \cdots ; z_n => P_n]$

Also, for a n-ary agent A defined as

$$A(x_1, \ldots, x_n) \overset{\text{def}}{=} P$$

we regard $A(y_1, \ldots, y_k)$ $k \leq n$ to be an agent of arity $n - k$ which is defined as agent $\hat{A}$

$$\hat{A}(x_{k+1}, \ldots, x_n) = P\{y_1/x_1, \ldots, y_k/x_k\}$$

An agent defining equation $A(y_1, \ldots, y_n) \overset{\text{def}}{=} P$, can represent recursion since P may contain any agent identifier, even A itself. We shall define, using recursion on agents a special process expression, called *replication*, represented as $!P$, by the equation,

$$!P \overset{\text{def}}{=} P \mid !P$$

In effect, $!P = P \mid P \mid \ldots$, indefinite copies of P in composition.

In fact, provided the number of recursive agents is finite, we can encode them by replications [Milner 1991]. In the discussion to follow we shall freely use replication as a variant of recursive agents.

1.4 Examples

1.4.1 Addition in π-calculus

Consider modeling an agent that represents a natural number in church numeral representation. An agent that represents a natural number n, $\underline{n}(s, z)$ communicates n times on a channel called the *successor* channel s and once on the *zero* channel z before becoming inactive. Thus

$$\underline{n}(s, z) \overset{\text{def}}{=} \overline{s}^n.\overline{z}.0$$

The behavior of the above process is illustrated in Figure 1 and 2. An addition process takes two natural numbers represented using the channels s_1, z_1 and s_2, z_2 and returns their sum as a natural number represented using channels s, z. The channels have been paired for notational convenience.

$$Add(s_1, z_1, s_2, z_2, s, z) \overset{\text{def}}{=} (s_1.\bar{s}.Add(s_1, z_1, s_2, z_2, s, z) \ + \ z_1.Copy(s_2, z_2, s, z))$$

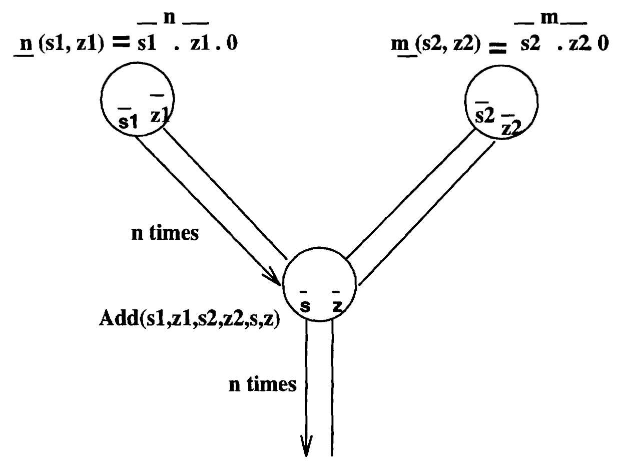

Figure 1: **Operand 1 communicates on successor channel**

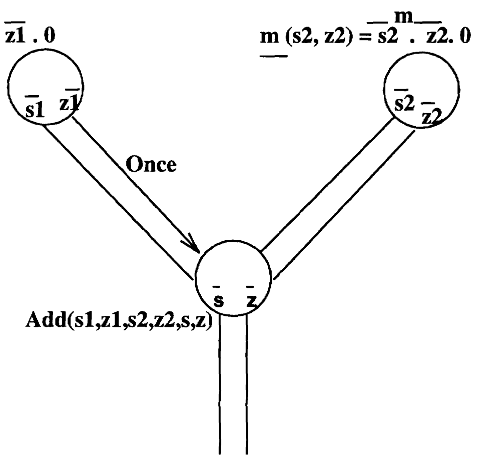

Figure 2: **Operand 1 communicates on zero channel**

The *Add* process signals on the successor channel corresponding to the output s while the first operand signals on its successor channel s_1. Once the operand signals on its zero channel, the *Add* agent copies the second operand on to the output channel pair s, z. This is illustrated in Figure 3 and 4. The *Copy* agent replicates the signal pattern on channels x and y on to channels u and v. It is defined as follows.

$$Copy(x,y,u,v) \overset{\text{def}}{=} (x.\overline{u}.Copy(x,y,u,v) + y.\overline{v}.0)$$

Thus the process $\underline{m+n}(s,z)$ is represented by

$$(\nu\ s_1\ z_1\ s_2\ z_2)(\underline{n}(s_1,z_1)\ \mid\ \underline{m}(s_2,z_2)\ \mid\ Add(s_1,z_1,s_2,z_2,s,z))$$

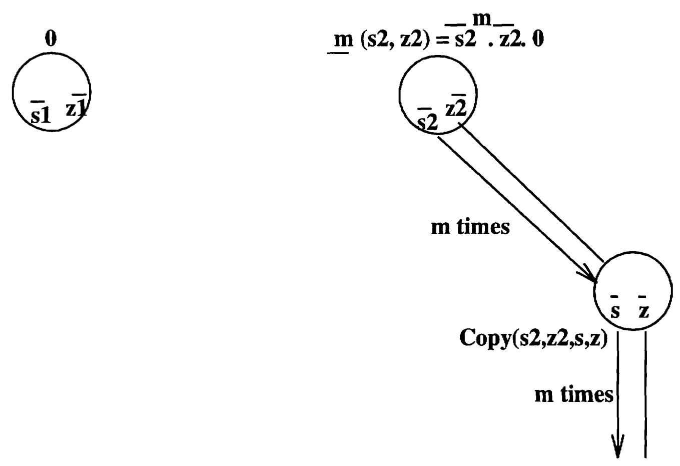

Figure 3: **Operand 2 communicates on successor channel**

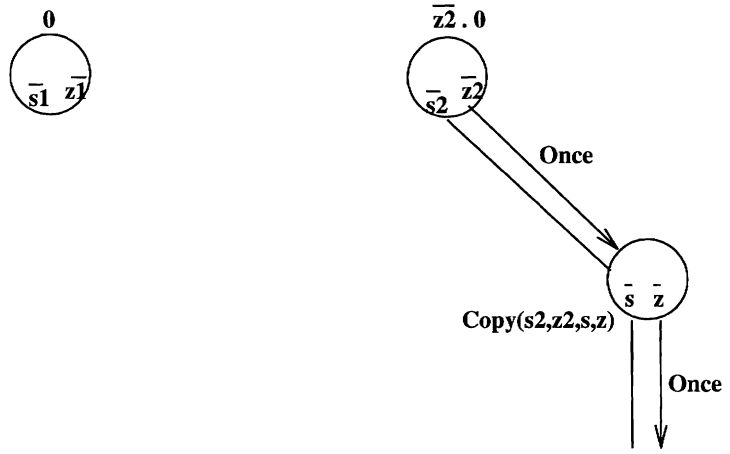

Figure 4: **Operand 2 communicates on zero channel**

1.4.2 Simulating Higher order π-calculus

π-calculus as defined here does not allow processes to be contents of communications. In this sense, it is first order in nature. However in an attempt towards modeling mobility of processes it is conceivable to define a higher order π-calculus [Milner 1991] in which processes are transmitted and received over channels as if they were data. The following example shows that the first order π-calculus is rich enough to simulate the effect of having processes as data, by using only names as data.

Consider extending the first order π-calculus to have agents transmit and receive processes. In the following example names in uppercase represent processes or process variables. Let

$$P(x) \stackrel{\text{def}}{=} \overline{x}R.P' \qquad \text{and} \qquad Q(x) \stackrel{\text{def}}{=} x(X).(X \mid Q')$$

After one interaction, $P(x) \mid Q(x)$ reduces to $P' \mid R \mid Q'$.

We can simulate the same effect in first order π-calculus by locating the "floating" agent at a new name z, and passing the address z to Q instead.

$$\hat{P}(x) \stackrel{\text{def}}{=} (\nu z)(\overline{x}z.(P' \mid z.R)) \qquad \text{and} \qquad \hat{Q}(x) \stackrel{\text{def}}{=} x(y).(\overline{y}.0 \mid Q')$$

$$\begin{aligned} \hat{P}(x) \mid \hat{Q}(x) &\equiv (\nu z)(\overline{x}z.(P' \mid z.R)) \mid x(y)(\overline{y}.0 \mid Q') \\ &\xrightarrow{\tau} (\nu z)((P' \mid z.R) \mid (\overline{z}.0 \mid Q')) \\ &\xrightarrow{\tau} P' \mid R \mid Q' \end{aligned}$$

as in the previous case.

It is noteworthy that higher order π-calculus provides a greater clarity of expression which is obscured in its simulation in first order π-calculus.

1.5 Equivalence of Processes

Some of the agents defined by the syntax above, are equivalent. Some of them may be identified simply based on their syntactic structure, while others are equivalent based on their interaction behavior.

1.5.1 Structural Congruence

Definition *Structural congruence* $\equiv$ is defined as the smallest congruence relation such that the following laws hold.

1. Agents are identified if they only differ in the names of their bound variables.
2. $P + 0 \equiv P,\ P + Q \equiv Q + P,\ (P + Q) + R \equiv P + (Q + R)$
3. $P \mid 0 \equiv P,\ P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
4. $(\nu x)0 \equiv 0, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
5. If $x \notin fv(P)$ then $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$
6. If $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ then $A(y_1, \ldots, y_n) \equiv P\{y_1/x_1, \ldots, y_n/x_n\}$

Structural congruence helps to factor out congruence induced by physical structure of agents from that induced by their interaction behavior. It also helps to bring the communicants to be neighbors as suggested in [Berry and Boudol 1990].

1.5.2 Simulation and Congruence

Two sequential programs are behaviorally equivalent if their input-output relationship is the same. But for concurrent programs the intermediate states that occur as the computation progresses are also important, since the intermediate states may be exploited by the environment to produce a different overall behavior. In the following example both programs P and Q replace the value of the variable x with 5. However when run in parallel with the program R each would produce different results.

- $P : x := 5;$
- $Q : x := 3; x := x + 2;$
- $R : x := 10;$

The behavior of a program is characterized by observing how it would communicate with an observer (such as program R in the above example). Two programs would be deemed to be congruent if they display identical behavior with every observer. This method of characterizing equivalence is called as *observational congruence.* Observational equivalence was introduced in [Ross 1985] and it captures behavioral equivalence between concurrent programs. Two observationally congruent programs may be used interchangeably in a system, without affecting the observational behavior of the system.

Definition *Simulation* is a binary relation $\Re$ over agents such that if $P \Re Q$ then

1. If $P \xrightarrow{\alpha} P'$ and α is a free action, then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P' \Re Q'$.
2. If $P \xrightarrow{x(y)} P'$ and $y \notin v(P) \cup v(Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all w $P'\{w/y\} \Re Q'\{w/y\}$
3. If $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin v(P) \cup v(Q)$, then for some Q', $Q \xrightarrow{\overline{x}(y)} Q'$ and $P' \Re Q'$.

- $\Re$ is a *bisimulation* if $\Re$ and its inverse are simulations. We shall denote the bisimulation relation by $\approx$.
- *Equivalence* of two processes $\dot{\sim}$ is the largest bisimulation. Thus two processes are equivalent if and only if they are related by some bisimulation.

The primary intuition behind simulation is that every transition of P can be simulated by a transition of Q such that the residue P' and Q' remain in the simulation. This is reflected in the first clause of the definition for simulation.

Secondly, to simulate an input action, it is not sufficient to require that the residues continue to be in the simulation. A process $x(y).P$ may receive any arbitrary name w and evolve into process $P'\{w/y\}$. To simulate such an action, it not only requires that the same action be performed, but also requires that the residues be in the simulation for all input values w. This intuition is captured by the second clause of the definition. The final clause is used for simulation of agents that perform a bound output action.

The following example demonstrates that bisimulation is not preserved under substitution of names.

$$F(x,y) \overset{\text{def}}{=} \overline{x} \mid y$$

$$G(x,y) \overset{\text{def}}{=} \overline{x}.y + y.\overline{x}$$

$$F(x,y) \dot{\sim} G(x,y) \qquad \text{but} \qquad F(x,x) \not\dot{\sim} G(x,x).$$

Observe that $F(x,y)$ and $G(x,y)$ have the same set of actions and hence are equivalent. However under the substitution $\sigma = \{x/x, x/y\}$, $F(x,x)$ can perform a silent action that $G(x,x)$ cannot perform. Hence they are not equivalent.

Definition *Congruence* Two processes P and Q are congruent, $P \sim Q$, if $P\sigma \dot{\sim} Q\sigma$, for all substitutions σ.

2 Use of π-calculus primitives to model programming concepts

In this section we shall take a closer look at the distinctive features of π-calculus with a view of determining their usefulness for representing programming concepts.

Some of the distinctive features of π-calculus are

- Names are treated as primitive entities.
- Channels are named instead of processes.
- Contents of channels are names and not processes.
- Restriction construct, (ν)

2.1 Expressing Client-Server Model of Computation

In a client-server model of computation, a server process provides a service upon request to a client process. The client names a service and passes the required parameters to the server. The server commences execution of the named service with the parameters provided by the client, while the client waits for results from the server. On completion of the service, the server transmits the results to the client, which resumes its execution. Such a model of computation can be expressed naturally in π-calculus.

The following example illustrates the client-server model of computation.

$$\begin{array}{rcl} P & \equiv & (\nu u)(\nu y)(\nu z)(\overline{x}u.\overline{u}y.u(z).P') \\ Q & \equiv & (\nu q)(x(v).v(p).R(p,v)) \\ S & \equiv & P \mid Q \end{array}$$

In the system S, the client P establishes a private communication channel u with the server Q which provides services determined by R (whose definition is not shown). The client then passes the parameters on the private channel to the server, which uses it as the input parameter for the service requested. Notice that the name of the private channel between the client and the server is passed on to R. This channel name is used to transmit the results of the service back to the client.

The example above is oversimplified, to illustrate, the idea without the minor details. A detailed illustration is provided in the translation of a client-server communication construct in the discussion of [Walker 1990] in the next section.

2.2 Naming of channels

The decision to name channels instead of processes, enhances the richness of expression of π-calculus to express the client-server model of computing. This feature allows a service to be associated with a channel name which can be used to model transparent access to server processes, in situations where more than one server process provides a service. Also, a process, by using different channels of interaction can bear different roles to different clients. This would not be possible to model if processes were to be named instead of channels. We conjecture that naming channels, would also help to overcome issues of unique naming of processes, a major issue in the distributed computing community. However a study of this is beyond the scope of the present discussion.

2.3 Restricting Scope of Names

Yet another uniqueness of π-calculus is the ν construct. The names restricted by the ν construct, are private to the process it prefixes. This can be used to model private resources of a process. Consider, the following processes

$$\begin{aligned} S_1 &\stackrel{\text{def}}{=} (\nu x)((\overline{x}.P \,|!x.Q) \mid \overline{x}.R) \\ S_2 &\stackrel{\text{def}}{=} ((\nu x)(\overline{x}.P \,|!x.Q)) \mid \overline{x}.R \end{aligned}$$

In process S_1, the resource Q may be accessed by both P and R. However in S_2, the resource Q is located on a private channel of P, which makes it a private resource, available only to P. R cannot access it. Selective access to Q may be provided, if S_2 is modified to S_3 as follows

$$S_3 \stackrel{\text{def}}{=} (\nu y)(((\nu x)(\overline{x}.P \,|!x.Q) \mid \overline{y}x) \mid \overline{x}.R \mid y(u).S)$$

Now, S may obtain access to the private channel of P on which Q is located through the name u. Communications on u would interact with the private resource of P.

In the above examples, let P, R and S be finite processes, that eventually terminate. Q, persists to exist because of replication. However, Q becomes inaccessible to all other processes because of the restriction. In the discussion to come, deadlocked processes are modeled in this manner.

π-calculus makes a unique decision by disallowing processes to be contents of communication. This restriction has a pleasant benefit. Passing processes over channels would provide unrestricted access to all resources that are private to a process. Passing names instead of processes helps to make selective resources of a process available for public usage. This serves as an excellent means to model restricted access to resources.

2.4 Modeling Control Strategies

Various control strategies can be modeled using π-calculus primitives. [Walker 1990] uses π-calculus to model sequential and iterative control strategy of an imperative language while [Ross 1990] uses it to model the left-to-right-depth-first strategy of Prolog. π-calculus communication viewed as synchronization signals is used for this purpose. A simple sequencing of processes is illustrated by the following example.

$$P \textit{ before } Q \quad \stackrel{\text{def}}{=} \quad (\nu x)(P(x) \mid x.Q) \quad x \notin v(P)$$

The execution of P and Q is sequenced such that Q waits until P completes execution. Upon completion of P a local synchronization signal is transmitted which triggers the execution of Q. Notice that the synchronization signal is passed as a parameter to P.

With the use of the sequencer the familiar iterative statement *while E do S* is modeled as follows. We assume the expression E to return constants TRUE or FALSE on evaluation. $[\ E]^\circ\ v$ is the process that represents the expression E. The value of the expression is returned on the channel v. The values TRUE and FALSE are represented by two special channel names $TRUE$ and $FALSE$.

$$W \quad \stackrel{\text{def}}{=} \quad (\nu v)([\ E]^\circ\ v \mid v : [TRUE => [\ S]^\circ\ \ before\, W; FALSE => 0])$$

The process works as follows. Upon evaluation of E, the process $[\ E]^\circ\ v$ outputs a $TRUE$ or $FALSE$ on the channel v. If the value is $TRUE$ then the process $[\ S]^\circ$ representing the statement S is executed before reexecuting W. If the output of $[\ E]^\circ\ v$ is $FALSE$ then the iteration terminates.

2.5 Representing Complex Data Objects

Complex data objects such as lists can be represented in π-calculus. We use the constructors $Cons$ and Nil to represent a list. For example the list [1,2] is represented as

$$Cons(1, Cons(2, Nil))$$

A list l is represented in π-calculus as an agent $[\ l]^\circ\ (x)$, where x serves as its access channel. We will assume two special channel names NIL and $CONS$ which will be treated as special values. The processes representing the $Cons$ cell and Nil are as follows.

$$[\ Nil]^\circ\ (x) \stackrel{\text{def}}{=} \overline{x}NIL$$
$$[\ Cons(v,l)]^\circ\ (x) \stackrel{\text{def}}{=} (\nu y)(\overline{x}Cons, v, y | [\ l]^\circ\ (y))$$

Observe that the data value v is ephemeral in that it disappears after an access. It needs to be copied back after every access just as a memory refresh operation at the address of the variable copies its value back to its location. Persistence of data values can be modeled by copying the data back to its location. Thus a persistent list is represented as follows.

$$[\ Nil]^\circ\ (x) \stackrel{\text{def}}{=} \overline{x}NIL \mid [\ Nil]^\circ\ x$$
$$[\ Cons(v,l)]^\circ\ (x) \stackrel{\text{def}}{=} (\nu y)(\overline{x}Cons, v, y \mid [\ l]^\circ\ y \mid [\ Cons(v,l)]^\circ\ (x))$$

The features discussed in this section orient π-calculus more towards object-oriented paradigm of computing, in which objects are named and pointers are used to provide access to one another. Data values are also represented as processes and instead of passing complex data objects as values, accesses to the processes representing them are passed as values. Thus π-calculus serves as an attempt to provide a canonical encapsulation of object-oriented paradigm of computing just as λ-calculus provides for functional paradigm. It also helps to formalize the operational semantics of such languages (which hither to fore have been treated mostly informally [Milner 1991]).

3 Translations of Programming Languages into π-calculus

Sequential programming languages can be broadly characterized into one of the following paradigms.

- Object-Oriented programming languages
- Logic programming languages
- Functional programming languages

Object-oriented programming languages are characterized by the presence of objects that interact by sending messages to each other. Notions of inheritance and data encapsulation are central to this paradigm. A variable in such languages is associated with a memory location whose value can be updated by the assignment operation. Computation proceeds in a sequential manner by retrieving and updating the values of program variables. The order of evaluation of statements and expressions crucially affects the results computed in such languages.

A program of a *logic programming* language consists of rules and facts. These languages are declarative in the sense that the rules and facts are specified by the programmer, and the language specifies the control mechanism to search the rule set. Computation is characterized by the solutions to queries about the rule set. Solutions to queries change depending on the search control strategy used. Unification is used to answer queries. Variables are used as place holders for values during the process of unification. They are set to a value when unification succeeds and are refreshed when unification fails. They are not associated with a fixed location as in imperative languages. Prolog is a representative of such a programming paradigm.

Functional programming is characterized as programming that uses function application as the *only* control structure. Binding of variables to terms (values or expressions) is used to associate a name to a value unlike the assignment statement of object-oriented languages. Variables can be bound only once but can be read from many times.

In what follows we present translations into π-calculus of three languages, L_2, Prolog and λ-calculus that represent the three paradigms of programming. The translations can be used for purposes, among others, to reason about properties of programs [Ross 1990] and to demonstrate the richness of π-calculus [Milner 1989].

3.1 An Object-Oriented Language

The paper [Walker 1990] provides a translation of two object-oriented languages L_1 and L_2 into π-calculus. A computation in these languages consists of a network of objects interacting with each other. The network topology dynamically changes as computation proceeds. The ability of π-calculus to model changing communicating network topology, is effectively used for this purpose. Also, a variable is modeled as a memory location using π-calculus.

We shall examine the syntax and capabilities of the language L_2 in the next section and follow it with a discussion of its translation into π-calculus. We will not discuss the language L_1 owing to space constraints.

3.1.1 Introduction to language L_2

A program of L_2 consists of a sequence of *class* declarations. Instances of the classes are called *objects.* The computational behavior of a system of objects governs the computational behavior of a program with a distinguished root object initiating the computation. An object consists of local variable declarations, a number of procedures or methods and a number of statements. Interactions among objects is for the purpose of method invocation and is done by means of rendezvous mechanism. Communication structure may evolve through the communications of references to new objects that can be created during computation.

Syntax of L_2

Let X, Y, Z be program variables. We shall limit our discussion to certain forms of expressions and statements only. For a complete repertoire the reader is referred to [Walker 1990]. The syntax for a few expressions is as follows.

$$\begin{array}{rcll} E & ::= & X & \\ & | & k & (k : nat \in N) \\ & | & E_1 + E_2 & \\ & | & E_1!M(E_2) & \\ & & \vdots & \end{array}$$

Each well-formed expression is one of *nat* or *ref* type. The syntax for some statements is as follows.

$$\begin{array}{rcl} S & ::= & X := E \\ & | & X := new_c \\ & | & answer(M_1, \ldots M_n) \\ & & \vdots \end{array}$$

Objects that are instances of a class C are created by the new_c construct. Reference to the object returned by new_c is assigned to a variable of type *ref.* A class declaration is of the form

$$Cdec ::= Class\ C\ is\ Vdec,\ Mdec\ in\ S$$

where $Mdec$ represents a method declaration.
A method declaration is of the form

$$Mdec ::= method\ M_1(X_1, Y_1)\ is\ S_1 \ldots M_n(X_n, Y_n)\ is\ S_n$$

with M_i distinct from one another. The X_is serve as the input parameter and the Y_is serve as output variables in which the result of the evaluation of S_is are stored.

The value returned by the expression $E = E_1!M(E_2)$ is the value returned by the object referred by E_1 after the invocation of the method M with parameter E_2. The method M is invoked if the object referred by E_1 executes the *answer* statement. The $answer(M_1, \ldots, M_k)$ statement invokes the method M_i for some i, with the parameter supplied by a client. A client is an object that is seeking to evaluate the expression $E_1!M_i(E_2)$ with E_1 a reference to the server, an object containing the answer statement. The execution of the client is blocked until the server returns a value. This construct helps to model the client-server mechanism of function invocation elegantly.

3.1.2 Translation of L_2 into π-calculus

The translation function $[\,]^\circ$ maps the constructs of L_2 to agents in π-calculus. We shall discuss only the salient points of the translation in [Walker 1990], in this report.

Translation of variables

A variable X is viewed as a memory location. Modeling the behavior of the memory location effectively serves as a translation of a variable declaration. The memory location contains the name of a link to an agent that represents the value of the variable. Each memory location for a variable X is associated with two constants [1] r_X and w_X that serve the purpose of its *read* and *write* channels. Thus the translation of the declaration of a variable of non-reference type is

$$
\begin{aligned}
[\ var\ X : t]^\circ &\overset{\text{def}}{=} Loc_X \\
Loc_X &\overset{\text{def}}{=} w_X(y).Reg_X(y) \\
Reg_X(y) &\overset{\text{def}}{=} \overline{r_X}y.Reg_X(y) + w_X(z).Reg_X(z)
\end{aligned}
$$

Assignment to a variable is done by storing the link to the agent that represents the new data value in the memory location, Loc_X, that represents the variable. Upon declaration of a variable all reads from it will result in the blocking of the reading process until an assignment stores a link name z in the location. Once an assignment is made, the variable is represented as a register $Reg_X(y)$ with y being the link to the agent representing the value of the variable.

Two other constants REF and NIL are used in the translation of a *ref* variable X. They signify respectively the state of X containing and not containing a reference to an object. An uninitialized reference variable contains a NIL. A read operation on an initialized reference variable communicates the flag REF and the name of the link to an agent that represents the value of the variable. Similarly a write operation is completed in two interactions. The following is the translation for reference variable declaration which is similar to that of a non-reference variable but for the two-step read and write operations.

[1] As discussed in [Milner and Walker 1989] constants can be replaced by special channel names

$$[\ var\ X : ref]^{\circ} \overset{\text{def}}{=} Loc_X$$

$$Loc_X \overset{\text{def}}{=} \overline{r_X}NIL + w_X[NIL => Loc_X; REF => w_X(y).Reg_X(y)]$$

$$Reg_X(y) \overset{\text{def}}{=} \overline{r_X}REF.\overline{r_X}y.Reg_X(y) + w_X[NIL => Loc_X; REF => w_X(z).Reg_X(z)]$$

Translation of Class construct

Every class C is associated with a π-calculus constant c which is the name of the link on which requests for creation of objects are accepted. Interaction on this channel results in the creation of a new link name and a copy of the body of the class. The new link name is private to the new object and serves as its channel for communication. The translation of a class declaration is

$$[class\ C\ is\ Vdec\ in\ S]^{\circ}\ (c) \overset{\text{def}}{=} \overline{c}(w).![\ Vdec\ in\ S]^{\circ}\ (w)$$

$$[\ Vdec\ in\ S]^{\circ}\ (w) \overset{\text{def}}{=} \nu r_{X_i}, w_{X_i}([\ Vdec]^{\circ}\ \mid [\ S]^{\circ}\ (w))(\forall X_i \in Vdec)$$

The replication operator in the above translation allows for creation of an indefinite number of objects that are instances of a class. Also new constants r_{X_i} and w_{X_i} are created for each variable X_i local to a class every time an object is created by the new_c expression.

Translation of Expressions

An expression is translated as an agent that communicates the value of the expression along a channel (value channel), as shown below. However such agents yield a value only once. They are ephemeral and cease to exist once the expression they represent is evaluated. We shall discuss two cases of expression translation for illustrative purposes.

$$[\ k]^{\circ}\ (v) \overset{\text{def}}{=} (\overline{v}ONE)^k.\overline{v}ZERO$$

$$[\ E_1 + E_2]^{\circ}\ (v) \overset{\text{def}}{=} (\nu v_1 v_2)([\ E_1]^{\circ}\ (v_1) \mid [\ E_2]^{\circ}\ (v_2) \mid [\ +]^{\circ}\ (v_1, v_2, v))$$

$$[\ +]^{\circ}\ (v_1, v_2, v) \overset{\text{def}}{=} v_1 : [ZERO => [+]^{\circ}\ '(v_2, v)\ ;\ ONE => \overline{v}ONE.[\ +]^{\circ}\ (v_1, v_2, v)]$$

$$[\ +]^{\circ}\ '(v_2, v) \overset{\text{def}}{=} v_2 : [ZERO => \overline{v}ZERO\ ;\ ONE => \overline{v}ONE.[\ +]^{\circ}\ '(v_2, v)]$$

where ONE and $ZERO$ are constants.

A natural number k, is represented by a process that communicates the unary representation of the number on its value channel. The flag $ZERO$ indicates the termination of the unary representation. Addition of two expression is a parallel composition of the translated operands along with a process that implements the addition operation. The two operands are evaluated with new and private links as their value channels. The private links are provided as parameters to the addition process. The addition process adds the numbers represented by the private channels and returns the unary representation of the result on the value channel of the expression.

Evaluation of an expression results in creation of a process that represents the value of the expression. Indefinite number of private instances of the agent representing the value of the evaluated expression may be obtained via a public channel that provides access to the process. Hence an assignment statement [$X := E]^\circ$, with X a variable of non-reference type, is translated to store the public channel to the process representing the value of E, in the memory location that represents X as illustrated in figure 5.

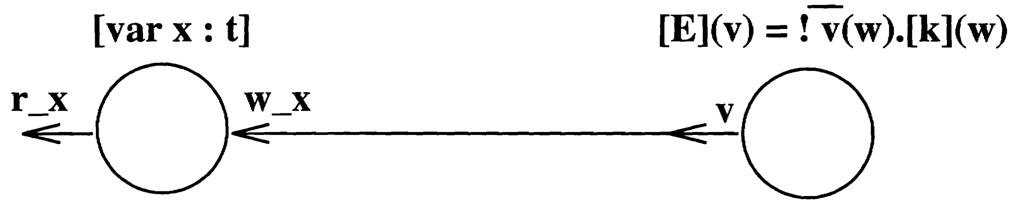

Figure 5: A value k stored in a variable x

The evaluation of a variable X, as a result of the translation for assignment, proceeds by

1. Accessing the link to the process representing the value. This is stored in the location that represents X, shown in figure 6.
2. Obtaining the link to a private instance of the process representing the value of X, shown in figure 7.
3. Interacting with the private instance to receive the value it represents, shown in figure 8.

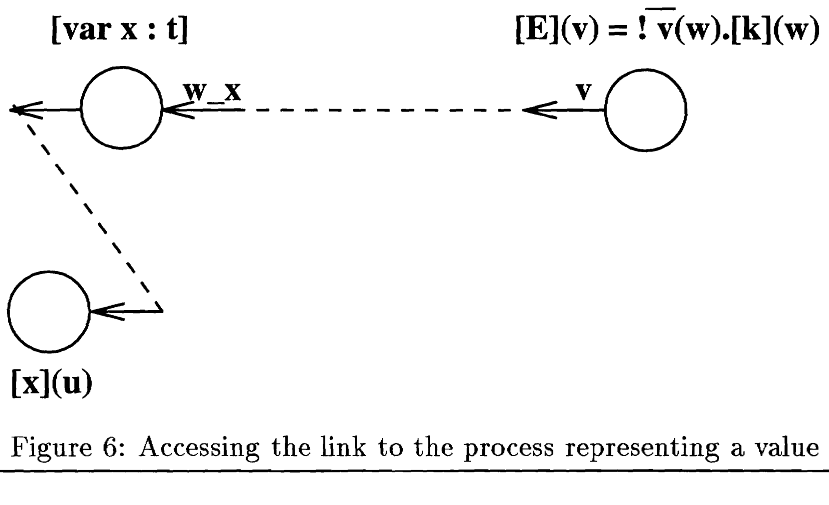

Figure 6: Accessing the link to the process representing a value

The assignment statement for the creation of an object using the new_c expression is translated as follows.

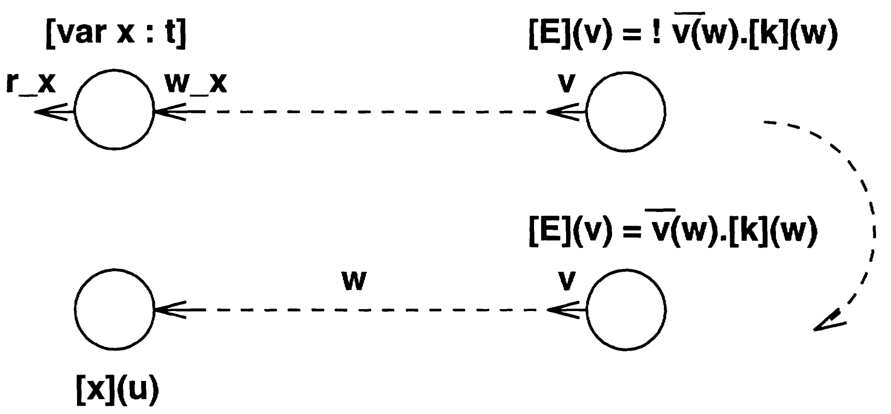
[var x : t]
$[E](v) = !\,\overline{v}(w).[k](w)$
r_x
w_x
v
$[E](v) = \overline{v}(w).[k](w)$
w
v
[x](u)

Figure 7: Obtaining the link to a private instance of the process

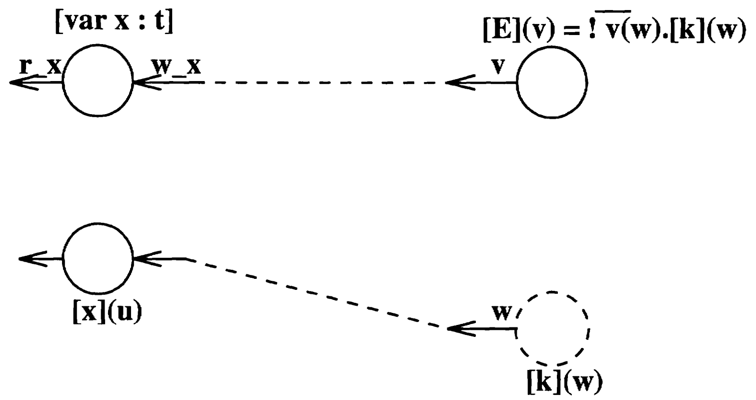

Figure 8: Interacting with the private instance to receive the value

$$[\, X := new_c]^{\circ}\ (w) \ \stackrel{\text{def}}{=}\ c(z).\overline{w_x}REF.\overline{w_x}z.Done$$

c refers to the π-calculus constant that is associated with every class. z would be the private link to the new object and is stored in Reg_X. *Done* is used to indicate the completion of the process.

Translation of Communication Constructs

Translation of the method construct along with the communication constructs are as follows.

At The Client End

$$[\![E_1!M(E_2)]\!]^{\circ}\,(v) \overset{\text{def}}{=} (v_3)((v_1)([\![E_1]\!]^{\circ}\,(v_1) \mid v_1(u)[u = REF]v_1(w).Done)$$
$$\quad before$$
$$((v_2)([\![E_2]\!]^{\circ}\,(v_2) \mid Eval(v_2, v_3))$$
$$\quad before$$
$$(u)\overline{w}u.\overline{u}m.\overline{u}v_3.u(v').v'(v'').Copy(v'', v)))$$

At The Server End

$$[\![answer(M_1, ..., M_n)]\!]^{\circ}\,(w) \overset{\text{def}}{=} w(u).u : [\, m_i => u(v).m_i(z).$$
$$\overline{z}w.\overline{z}v.z(v').\overline{u}v'.Done]\{i = 1, k\}$$

$$[\![method\ M(X,Y)\ is\ S]\!]^{\circ} \overset{\text{def}}{=} \overline{m}(z)!M(z)$$
$$M(z) \overset{\text{def}}{=} (N)(Loc_X \mid Loc_Y \mid z(w).z(x).\overline{w_X}x.([\![S]\!]^{\circ}\,(w)$$
$$before\ r_Y(v').\overline{z}v'))$$

The start of a communication begins with the client evaluating the expression $E_1!M(E_2)$. The expression E_1 is evaluated to a link w to the server (object referred to, if it exists). Then the expression E_2 is evaluated and a private link v_3 is used as a channel to the process representing the value of E_2. At this point the client remains suspended until the server executes an answer statement. Then a private link u is communicated to the server via the channel w. Along the private channel u, the reference to the method requested for execution m is then communicated. If the server offers the method requested (m_i), for public usage, then the link to the argument for the method, v_3 is received along u . The server then requests from the agent representing the declaration of that method for a private copy of the agent representing the body of the method. The private channel z is used by the server to communicate with the private copy of the body of the method m_i. Along z the server communicates w, the reference to the current object and v, the name of the link to the parameter. The agent representing the body of the method stores the input parameter in local Reg_X and proceeds with the execution of the statement S. On completion of S, the name stored in local Reg_Y is returned to the server as the link to the process representing the output value of the method. The server in turn returns this name to the $E_1!M(E_2)$ along the private link u, and then indicates completion. Finally a link to the private copy of the output value is obtained and the value is copied using the *Copy* agent.

The translation of the communication construct makes several assumptions.

1. Only one client may interact with a server. If several clients are waiting to be serviced, one of them is picked nondeterministically and served.

2. The method requested by the client must be offered by the server. If it is not then the server fails to proceed and both the server and client are deadlocked.

3. At the time a server executes an answer statement, at least one client needs to be waiting for service. If there are no clients in the system, the server is blocked.

The rest of the constructions are translated in an obvious manner and will not be discussed here.

3.1.3 Remarks

[Walker 1990] serves to demonstrate the expressive power of π-calculus by encoding two languages that have communication constructs. However the paper does not however justify the choice of the two languages as being representative of the object-oriented paradigm. It fails to demonstrate how the language constructs capture concepts such as encapsulation and inheritance that are central to the object oriented paradigm.

Also, the translation of expressions of type *nat*, involve an infinite set of recursive processes. This could be avoided by using the π-calculus with abstractions and concretions. We made an attempt to transform the translation using this version of π-calculus. However due to the delicate relationships embedded in the translation, such a transformation would require a complete overhaul of the current translation.

[Walker 1990] makes a point that by providing the π-calculus semantics it has avoided the use of sophisticated mathematical machinery required to establish the well-definedness of the semantics. However it does not include any proof of correctness of the translation nor examines its properties. The paper may be regarded as one of several possible variations of translating the languages L_1 and L_2 into π-calculus.

3.2 π-calculus semantics for Prolog programs

[Ross 1990] provides a process model interpretation of logic program computation that can be modeled using a process algebra. CCS has been chosen to be the process algebra to model such an interpretation. We shall present the same translation in π-calculus. Modeling logic programs as a system of processes has several advantages. Properties of logic programs may be proved using properties of processes that represent them. [Ross 1990] uses the notion of bisimilarity of processes to prove termination properties, to validate partial evaluation and source-to-source transformation of logic programs.

AND/OR trees define the declarative semantics of logic programs. They represent logical dependencies in computation and inference. The AND node requires that *all* its children (goals) be solved while the OR node requires that *one* of its children (clause) be solved. Operational semantics of logic programs may be provided by viewing the AND and OR nodes of an AND/OR tree as processes. Prolog uses a depth-first-left-right search strategy in solving for a clause. The clauses are searched in the textual order as they appear in the program. The control strategy is further affected by the use of control operators such as *cut*. The AND and OR agents of the AND/OR tree that represents a Prolog program determine the manner in which the tree is explored, in effect modeling a control strategy. Such an approach has been adopted by [Conery and Kibler 1985] for providing operational semantics for concurrent logic languages where the nodes have been treated as concurrent agents. [Ross 1990] uses sequential AND and OR agents instead.

3.2.1 Translation

Two special events, *succ*(θ) and *done* representing successful and unsuccessful termination signals, control the execution of a program. θ in *succ*(θ) represents a substitution of the form $X_i \leftarrow t_i$ returned by an unification agent upon successful termination. An empty substitution is represented as ϵ.

A finite computation may be viewed as a series of *succ* actions followed by a *done* action. A failure computation is represented by termination without any success actions. Thus a *succ* action is the primary means for observing computation and every terminating computation eventually results in a *done* action.

We shall discuss the translation provided in [Ross 1990] using π-calculus. The following is a translation of each construct of Prolog in π-calculus.

A construct of Prolog is represented as an agent in π-calculus. The agent is parameterized by two channels, s and d that represent the two special events *succ* and *done*. Output on s channel indicates a *succ* action while that on d channel indicates a *done* action.

1. Some definitions

   $Done(s,d) \overset{\text{def}}{=} \overline{d}$

   $True(s,d) \overset{\text{def}}{=} \overline{s}(\epsilon).\overline{d}$

2. Predicates (OR agents)

   $[\ P_1, P_2, \ldots P_n]^{\circ}\ (s,d)\ = P(s,d) \overset{\text{def}}{=} (P_1 \hat{;} P_2 \hat{;} \ldots \hat{;} P_n)(s,d)$

3. Clauses (AND agents)

   $[\ P_i : -G_1, \ldots G_n]^{\circ}\ (s,d)\ = P_i(s,d) \overset{\text{def}}{=} ([\ G_1]^{\circ} \ \triangleright \ldots \triangleright [\ G_n]^{\circ}\ )(s,d)$

4. Program Queries

   $[\ : -G_1, \ldots G_n]^{\circ}\ (s,d)\ = ([\ G_1]^{\circ} \ \triangleright \ldots \triangleright [\ G_n]^{\circ}\ )(s,d)$

5. Sequencing operator

   $(P \hat{;} Q)(s,d) \overset{\text{def}}{=} (\nu d')(P(s,d') \ | \ d'.Q(s,d))$

6. Goal backtracking operator

   $(P \ \triangleright Q)(s,d) \overset{\text{def}}{=} (\nu s'\ d'\ loc)\,(P(s',d') \ | \ NextGoal(s',d',loc,s,d)\,|!loc(s_1,d_1).Q(s_1,d_1))$

   $NextGoal(s',d',loc,s,d) \overset{\text{def}}{=} s'.(R(loc)\,\hat{;}\,NextGoal(s',d',loc))(s,d) + d'.Done(s,d)$

   $R(loc,s,d) \overset{\text{def}}{=} \overline{loc}\ s,\ d.0$

7. Single goal calls

   $$[\ G]^{\circ}\ (s,d) \overset{\text{def}}{=} \begin{cases} G(s,d)\ : G\ is\ a\ defined\ predicate \\ \overline{s}.Done(s,d) + Done(s,d) \quad : G\ is\ a\ builtin\ atom \\ Done(s,d) \quad : G\ not\ defined \end{cases}$$

8. Builtin Unification Agent

   $(\tilde{t_1} = \tilde{t_2}\ )(s,d) \overset{\text{def}}{=} \overline{s}(\theta).Done(s,d)\ +\ Done(s,d)$

The order for searching clauses is modeled by the OR agent that sequentially invokes the clauses in the order they appear in the text of the program. The special operator $\hat{;}$ uses a private channel to ensure the sequential order of execution of successive clauses.

Each clause is represented by an AND agent. It repeatedly resolves the goals in the clause body in a left-right order. A special operator $\triangleright$ is used to model the backtracking behavior of goals. In $(P \triangleright Q)(s, d)$ the goal P is invoked with private channels s' and d' to represent its termination nature. After the first solution of P, indicated by a signal on s', the goal Q is invoked. The invocation of Q is by the trigger mechanism that was used to simulate a higher order process in the example on page 9. New instances of the goal Q are created for each solution of goal P and are passed the channels s and d as parameters using the $R(loc, s, d)$ process. The sequence operator ensures that the next solution of P is processed once all the solutions of Q have been processed. This continues until P terminates. It is to be observed that after the first solution of P, both P and Q continue execution in parallel.

The unification agent is not modeled in [Ross 1990]. The paper assumes a builtin unifier agent that returns the substitution if unification of the terms succeeds. The builtin unification agent returns θ, the most general unifier of $\widetilde{t}_1$ and $\widetilde{t}_2$ on successful unification. If not it performs the termination action.

A Prolog program and its π-calculus translation is provided in figure 9.

$$\begin{array}{llll}
 & [\ a(X)]^{\circ}\ (s,d) & \stackrel{\text{def}}{=} & (a_1(X)\ \hat{;}\ a_2(X))(s,d) \\
a(1). & [\ a_1(X)]^{\circ}\ (s,d) & \stackrel{\text{def}}{=} & (X = 1)(s,d) \\
a(2)\ \ :-\ \ a(2).\ \Leftrightarrow & [\ a_2(X)]^{\circ}\ (s,d) & \stackrel{\text{def}}{=} & ((X = 2)\ \triangleright\ a(2))(s,d)
\end{array}$$

Figure 9: A logic program and its π-calculus translation

Having obtained the π-calculus representation of a Prolog program, the operational semantics of π-calculus may be used to reason about the execution of the program. Since the purpose of the translation is to observe the computational behavioral properties of Prolog programs, the behavior of the control operators $\hat{;}$ and $\triangleright$ can be provided at a higher level using bisimilarities. This entails providing bisimilar agents that represent all possible states of sequencing and backtracking mechanism. A new operator $\hat{\triangleright}$ is introduced to represent an intermediate state of $(P \triangleright Q)(s, d)$. This intermediate state corresponds to the situation when P produces a signal on s channel and the computation of Q is to commence using the substitution θ resulting from P.

$$(P \,\hat{\triangleright}\, Q)(s,d) \quad \stackrel{\text{def}}{=} \quad (\nu s'\ d')(P(s',d')\ \mid\ s'.(Q; NextGoal)(s,d))$$

Bisimilarities that may be used as rewrite rules are as follows.

$$\begin{array}{lcll}
\textbf{Seq} & : & (Done \overset{\circ}{;} P)(s,d) & \approx \ P(s,d) \\
\textbf{Back}-\textbf{1} & : & ((\overline{s}(\theta).P) \ \triangleright \ Q)(s,d) & \approx \ (P \,\hat{\triangleright}\, Q\theta)(s,d) \\
\textbf{Back}-\textbf{2} & : & (Done \ \triangleright \ Q)(s,d) & \approx \ Done(s,d) \\
\textbf{Back}-\textbf{3} & : & (P \ \hat{\triangleright} \ (\overline{s}(\theta).Q))(s,d) & \approx \ (\overline{s}(\theta).(P \,\hat{\triangleright}\, Q))(s,d) \\
\textbf{Back}-\textbf{4} & : & (P \,\hat{\triangleright}\ Done)(s,d) & \approx \ (P \ \triangleright \ Q)(s,d) \\
\textbf{Back}-\textbf{5} & : & (\overline{s}(\theta).P \ \triangleright \ Q)(s,d) & \approx \ (Q\theta \overset{\circ}{;} (P \triangleright Q))(s,d)
\end{array}$$

Resolution rule is a bisimilarity used to model a single resolution step by applying the substitution obtained on unification of the head of a clause and a goal, to the body of that clause.

Resol:

$$P_i(\tilde{t}\ )(s,d) \ \approx \begin{cases} Done(s,d) \ : \tilde{t} \quad and \ \ \tilde{t_i} \quad do\, not\ unify\ for\ any\ P_i \\[1em] \overline{s}(\theta).Done(s,d) \ : \theta = mgu(\tilde{t}\ , \tilde{t_i}\ )\ and \\ \qquad\qquad P_i(\tilde{x}\ )(s,d) \ \overset{\text{def}}{=} \ (\tilde{x} = \tilde{t_i}\ )(s,d) \\[1em] (Q\theta)(s,d) \ : \theta = mgu(\tilde{t}\ , \tilde{t_i}\ )\ and \\ \qquad\qquad P_i(\tilde{x}\ )(s,d) \ \overset{\text{def}}{=} \ ((\tilde{x} = \tilde{t_i}\ ) \triangleright Q)(s,d) \end{cases}$$

Translation of Cut

The cut control operator affects Prolog's search strategy. When a cut is invoked the following events occur

1. the choice points of the goals found prior to the cut in the clause are discarded,
2. the clauses following the clause with the cut are not searched.

These events are modeled in π-calculus by suspending agents. Agents are forced to deadlock. We define the operators that model cut as follows.

$$\begin{array}{lcl}
(A \ \overline{\triangleright} \ B)(s,d) & \overset{\text{def}}{=} & (\nu s'\ d')(A(s',d') \ | \ \ s'.B(s,d) \,|\, done'.Done(s,d))) \\
(A \ \overline{\triangleright}_c \ B)(succ, done, c) & \overset{\text{def}}{=} & (\nu s'\ d')(A(s',d') \ | \ \ s'.B(s,d) \,|\, d'.Done(s,c)) \\
(P \ \overset{\circ}{;} \ Q)(s,d) & \overset{\text{def}}{=} & (\nu c)(P(s,d,c) \ | \ \ c.Q(s,d))
\end{array}$$

The two operators $\overline{\triangleright}_c$ and $\overline{\triangleright}$ are used to model the first cut in the clause and the subsequent cuts respectively. It is evident that the first cut prunes the clause space while the subsequent cuts do not prune the clause search space. In the definition of $A \,\overline{\triangleright}\, B$ the first solution from A is used to invoke B. Subsequent solutions of A are ignored. This differs from the $\triangleright$ where all solutions of A are retrieved. The operator $\overline{\triangleright}_c$ terminates with either a signal *done* or c depending on whether the cut was invoked or not. The sequencing of a clause with a clause that has a cut is represented by the $\overset{\circ}{;}$ operator. In $P \overset{\circ}{;} Q$ the operator invokes Q only when the signal from P is c which means that the cut in P was not activated. Owing to this signaling mechanism, the agent that represents a clause with a cut, needs three channels to communicate, while the agent that represents a clause with no cut will need only two channels. The following bisimilarities apply to the operators that model cut. The bisimilarities describe the operational effects of cut which prunes goal and clause search.

$$\textbf{Cut}-\textbf{1} \;:\; ((\overline{s}(\theta).A \,\overline{\triangleright}_c\, B) \overset{\circ}{;} C)(s,d) \quad\approx\quad (B\theta)(s,d)$$

$$\textbf{Cut}-\textbf{2} \;:\; ((\overline{done}.A \,\overline{\triangleright}_c\, B) \overset{\circ}{;} C)(s,d) \quad\approx\quad C(s,d)$$

Appendix 4 illustrates the use of bismilarities between the Prolog program representation in π-calculus, for validating source-to-source transformation and proving termination of programs.

3.2.2 Remarks

[Ross 1990] presents the operational semantics of Prolog using CCS, the essence of which has been preserved and presented in π-calculus framework here. Prolog's search control strategy, with and without cut, is modeled by the synchronization mechanism provided by π-calculus. The behavior of a Prolog program is studied using its semantic representation which in this case are processes of π-calculus. Properties of programs can be proved using the notion of bisimilarity of processes. Equivalence between programs is established by demonstrating bisimilarity between their process representations.

[Ross 1990] assumes the presence of a unification agent and uses its behavior implicitly in proving properties of programs. The representation of the substitution returned by the unifier agent and the bookkeeping regarding binding of variables have not been worked out. In fact, the translation presented in [Ross 1990], may be regarded as a translation of a propositional logic language, since the translation of variables is not dealt with at all.

The translation provided in [Ross 1990] requires an indexed set of processes, $NextGoal_i$, indexed by pairs of goals, to simulate the backtracking of goals. Our translation to π-calculus avoids this by making $NextGoal$ simulate a higher order process.

3.3 Functions as Processes

The paper [Milner 1989] exhibits accurate encodings of two variants of λ-calculus in first order π-calculus. It may seem surprising at first sight to note that λ-calculus, in which variables can be bound to terms, can be simulated by first order π-calculus in which variables can only be bound to names. However upon reflection it is not so surprising since all implementations of functional languages on conventional machines work by passing values and addresses between registers, instead of passing complex entities such as functions. Hence there must be some way of encoding λ-calculus using π-calculus. [Milner 1989] demonstrates a translation that preserves the notion of reduction in λ-calculus. A reduction step of a λ-calculus term is simulated by atmost a short sequence of reductions in π-calculus.

The paper [Milner 1989] also attempts to define a precongruence relation on the translations of λ-terms that are related by a precongruence relation called *applicative simulation.*

3.3.1 Lazy λ-calculus

Syntax

Let $x, y, \ldots$ range over an infinite set of variables $\mathcal{X}$. Let $L, M, N \ldots$ range over the set of terms $\mathcal{L}$ of λ-calculus, which are defined as follows.

$$M ::= x \mid (\lambda x.M) \mid MN$$

The last two terms are called *abstraction* and *application* respectively. The familiar reduction relation takes the form

$$\beta : (\lambda x.M)N \rightarrow M\{N/x\}$$

where $\{N/x\}$ represents substitution of the term N for x.

Versions of the reduction relation differ by the context in which they admit β-reduction. Lazy λ-calculus admits β-reduction only at the extreme left of a term. This is called *lazy reduction.* Thus the term N is substituted as is, in the term M.

The lazy reduction relation $\rightarrow$ over $\mathcal{L}$ is the smallest relation which satisfies β-reduction along with the following rule for application.

$$APPL \quad : \quad \frac{M \rightarrow M'}{MN \rightarrow M'N}$$

Notice that given a term M, there is exactly one and only one term it can reduce to. Thus the lazy reduction relation $\rightarrow$ is determinate. We represent the reflexive transitive closure of $\rightarrow$ by $\overset{*}{\rightarrow}$.

We proceed to encode the terms of lazy λ-calculus into π-calculus. λ-calculus seems to be a basic calculus and one would expect the encoding to be simple if not obvious. However, the only term rewriting rule, β-reduction, uses a complex operation of substitution as a primitive. An encoding of λ-calculus, then needs to model substitution in π-calculus.

The approach adopted to simulate substitution in [Milner 1989] is to encode the environment model of term evaluation in π-calculus. In an environment model of term evaluation, terms are evaluated in an environment which binds variables to terms. In the term $M\{N/x\}$, the effect of substitution is achieved by evaluating the term M in an environment that binds the variable x to the term N. [Milner 1989] formalizes the environment model of term evalutation to achieve the encoding.

Translation of lazy λ-calculus

A term M of the λ-calculus is encoded as an agent $[\![M]\!]^{\circ}\, u$. The name u is the link (access channel) along which $[\![M]\!]^{\circ}$ receives its arguments.

If M is used as an argument to a function in a term and is bound to a variable x, then each occurrence of x in the scope of the binding would result in a computation of M. Hence each time M is called it must be told by the caller, where it can receive its arguments from. Thus an environment entry that binds variable x to a term M is translated as

$$[x := M] \stackrel{\text{def}}{=} !x(w).[\,M]^{\circ}\ w$$

w is the link along which the processing representing the term M receives its arguments. The caller of M provides this name along the link x, which is the bound variable in the environment. The agent M is called whenever the variable x appearing in a term N is evaluated. The replication operator in the translation is needed since the variable x may appear more than once in a term N and each evaluation of x would result in a call to M.

The translations of terms are as follows.

$$\begin{array}{lcl} [\,\lambda x.M]^{\circ}\ u & \stackrel{\text{def}}{=} & u(x,w).[\,M]^{\circ}\ w \\ [\,x]^{\circ}\ u & \stackrel{\text{def}}{=} & \overline{x}u \\ [\,MN]^{\circ}\ u & \stackrel{\text{def}}{=} & (\nu v)([\,M]^{\circ}\ v \mid \overline{v}(x), u.[x := N]) \quad (x \textit{ not free in } N) \end{array}$$

A lambda abstraction $(\lambda x.M)$ is represented by an agent that receives on u, the name of the actual parameter for x and a name w, the link along which M will receive its arguments. Note the use of the abstracted variable x of λ-calculus as a channel name in π-calculus. During communication, x would be bound to the input value on channel u. As a result, all occurrences of x in M would be substituted by the name that x would be bound to, simply by the operational semantics of π-calculus.

The translation of a variable x is an agent that transmits its access channel u, on x. Now, u serves as the access channel to $[\,N]^{\circ}$, the agent to which x is bound to, in the environment. Thus $[\,N]^{\circ}$ is located at the site of evaluation of x. For example, assume that x is bound to N in the current environment. Then evaluation of x is represented as follows.

$$\begin{array}{rcl} (\nu u)([\,x]^{\circ}\ u \mid [x := N]) & \equiv & \overline{x}u \mid !x(w).[\,N]^{\circ}\ w \\ & \longrightarrow & (\nu u)([\,N]^{\circ}\ u \mid !x(w).[\,N]^{\circ}\ w) \end{array}$$

In the translation for function application, M and its actual arguments are at the two ends of a private communication link, v. The agent $[\,N]^{\circ}$ will not be activated until $[\,M]^{\circ}\ v$ reduces to an abstraction $[\,\lambda y.M']^{\circ}\ v$ and is ready for its arguments on v. The double prefix of $[\,N]^{\circ}$ is the essence of lazy evaluation. The actual argument N is evaluated as many times as y appears in M'. The replication in the environment entry serves to provide as many copies of N as there are occurrences of y in M. After all the occurrences are evaluated, the environment entry for the private variable x, serves no purpose and the corresponding process would be deadlocked, since there would be no process to interact with it. This represents the garbage collection of an environment entry.

3.3.2 Supplementing the reduction relation

[Milner 1989] verifies the correctness of this translation by proving a theorem to the effect that the translation presented faithfully preserves the convergent and divergent properties of terms of λ-calculus. A reduction of a λ-calculus term is simulated by atmost a sequence of reductions by its π-calculus representation.

The reduction relation $\rightarrow$ used in the above example describes only the *intra-action* in a π-term. However a π-term not only performs intra-actions but can potentially interact with other π-terms of an environment. In case of lazy λ-calculus, interaction of a term with the environment cannot occur until the term reduces to an abstraction. There is a well-defined ordering of reductions and interactions for terms of lazy λ-calculus. But in π-calculus, owing to concurrency, intra-action and interaction can intermingle and this could change the behavior of a π-term in a context.

A translation of λ-terms must then concern itself with faithfully simulating the behavior of the term in a context. To be precise, the translation must preserve the distinguishability of λ-terms under observation. Abramsky defined a preorder $\lesssim$ called *application simulation* that relates two terms based on their convergence behavior.

Definition M *converges* to M', written as $M \downarrow M'$, if $M \xrightarrow{*} M' \not\rightarrow$; also, $M \downarrow$ if M converges to some M'.

Let $\mathcal{L}^0$ be the set of closed terms. It can easily be seen that if $M \in \mathcal{L}^0$ and $M \downarrow M'$ then M' is an abstraction.

Definition Let $L, M \in \mathcal{L}^0$, then $L \lesssim M$ if for all sequences of closed terms $\tilde{N}$, if $L\ \tilde{N}$ converges implies $M\ \tilde{N}$ converges.

Furthermore if $L, M \in \mathcal{L}$ have free variables $\tilde{x}$ we define $L \lesssim M$ if $L\{\tilde{N} / \tilde{x}\} \lesssim M\{\tilde{N} / \tilde{x}\}$ for all $\tilde{N} \in \mathcal{L}$.

Also $\lesssim$ is a precongruence, since M $\lesssim$ N, iff for every closed context $C[\_]$; a term with a single hole; if $C[M] \downarrow$ implies $C[N] \downarrow$.

The paper [Milner 1989] attempts to define a precongruence relation $\sqsubseteq$ between the translations of two λ-terms that preserves applicative simulation relation $\lesssim$ of the two terms. It succeeds in defining a precongruence over π-terms that preserves the following bi-implication only in the forward direction.

$$[\ M]^{\circ}\ u \sqsubseteq [\ N]^{\circ}\ u \Longleftrightarrow M \lesssim N$$

3.3.3 Call-by-value λ-calculus

In the call-by-value version of λ-calculus, a reduction in a closed term occurs only when the argument is an abstraction. The terms $\mathcal{L}$ are as before, but it is convenient to define terms that can serve as values as

$$V ::= x \mid \lambda x.M$$

Corresponding to lazy reduction relation of lazy λ-calculus, we have call-by-value reduction relation which is defined as follows.

Definition The call-by-value reduction relation $\rightarrow_v$ is the smallest relation that obeys the following rules.

$$\beta_v : \qquad (\lambda x.M)V \rightarrow_v M\{V/x\}$$

$$APPL : \qquad \frac{M \rightarrow_v M'}{MN \rightarrow_v M'N}$$

$$APPR : \qquad \frac{N \rightarrow_v N'}{MN \rightarrow_v MN'}$$

The two application rules allow reductions to take place at any location in a term, save that the argument in an application must be a value for it to qualify as a redex.

Translation

We proceed to provide a translation of the call-by-value λ-calculus into π-calculus. A λ-term M is thought of as an agent $[\ M]^\circ\ p$. However the channel p plays a dual role.

1. As before, it is used as a channel for the agent to receive argument values.
2. In addition to that, it is also used to signal that the term M has reduced to a value.

The signaling mechanism is required since M may occur as an argument of an application term and reduction is admitted only when M is a value. This constraint implies that abstracted variables of call-by-value λ-calculus are bound only to values. Hence the entries in the environment contain only values and their translations are

$$[y := x] \overset{\text{def}}{=} \ !y(w).\overline{x}w.$$

$$[y := \lambda x.M] \overset{\text{def}}{=} \ !y(w).w(x,p).[\ M]^\circ\ p$$

The environment entries do not differ significantly from those of the lazy λ-calculus, except for the unfolding of the translation for the environment variable. The translations for the terms are as follows.

$$[\ V]^\circ\ p \overset{\text{def}}{=} \overline{p}(y).[y := V]$$

$$[\ MN]^\circ\ p \overset{\text{def}}{=} (\nu q, r)\,(ap(p,q,r) \mid [\ M]^\circ\ q \mid [\ N]^\circ\ r)$$

$$ap(p,q,r) \overset{\text{def}}{=} q(y).\overline{y}(v).r(z).\overline{v}zp$$

A term that can be a value translates to an agent that announces its valuehood and provides access to a newly created environment entry that binds the value to a new variable y. This is distinct from the lazy λ-calculus case, in which an environment entry is created only when an abstracted variable is to be bound to a term.

Application of terms is translated as follows.

1. Two new links are used to serve as access channels to processes representing M and N which execute concurrently. Let y and z be the environment variables bound to the values of M and N. Further, let the value of M be $\lambda x.M'$.

2. The *ap* agent receives the name y on q, and along y transmits a new link name v, that would contain the channel names z (channel that provides access to the result of evaluating N) and p (the location at which the result of the application is required).

It is evident that in general $\rightarrow_v$ is not determinate, since a term may reduce to two different terms depending on which redex is reduced. However it is well known that if a term M reduces to M' and M'' then there exists a term N to which M' and M'' both reduce. Thus if $M \downarrow N$ then N is unique and so all reduction sequences terminate. [Milner 1989] proves that the translation provided for call-by-value λ-calculus faithfully preserves the convergent and divergent properties of terms. Also an attempt is made to define a precongruence that is equivalent to $\lesssim_v$, a precongruence similar to $\lesssim$

3.3.4 Remarks

[Milner 1989] serves to demonstrate that the function-argument form of computing is a special case of concurrent computing. It translates two versions of λ-calculus into π-calculus faithfully preserving the notion of reduction. A reduction of λ-calculus has been proved to correspond to a short sequence of reductions in π-calculus. Substitution of terms for variables in an application is simulated using the environment model of evaluation. The translation is so compact that it verges on incomprehensibility. The paper does little in terms of explanation of the translation to clarify the translation.

We shall compare the representation of a variable of λ-calculus with that of an object-oriented language, L_2.

1. A variable of λ-calculus is represented by a channel in π-calculus, while that of L_2 is represented by a process that represents a memory location for the variable.

2. A value (term) to a variable in λ-calculus is assigned on creation of an environment entry. It represents a process that has an effect of textual substitution of the value (term) of the variable at the location of evaluation of the variable.

   A value to a variable of L_2 is assigned by storing the access channel of the process that represents its value in the location for the variable. Evaluation of a variable would proceed to first access the access channel of the process that represents the value of the variable, obtain a private copy of the process, and copy value it represents to the location of evaluation of the variable.

3. Repeated assignments to a variable of L_2 may be done by updating the contents of the location that represents the variable to store the name of the access channel to the process that represents the new value. This operation does not have a counterpart in λ-calculus.

4 Conclusions

In this paper we have presented translations into π-calculus of languages that represent three differing paradigms of computing: object-oriented, logic and functional paradigm. The similarities of the translations are as follows.

1. Data values and expressions are represented as processes with access channels.

2. Data flow is modeled by passing the access channel of the process that represents the data, as a value.

3. Control flow is modeled using synchronization signals.

The ability to model control flow independently of the data flow, in π-calculus makes it a unifying framework for the three differing programming paradigms.

π-calculus may be used to model the major components of a conventional computer. In fact the translation given in [Walker 1990] treats memory to be a process. The control unit can be simulated by synchronization signals a primitive that is provided implicitly in π-calculus. The arithmetic and logic unit can modeled as a set of processes that model the basic operations of the unit.An example of an addition process was discussed earlier in this paper. In the light of the above discussion, it does not seem surprising that π-calculus serves as a unifying framework for the three languages. Although the unification of the languages takes place almost at the machine level, the operational semantics of the languages is still compositional in nature.

π-calculus with its ability to model the client-server computation shows promise to be a calculus that provides a canonical encapsulation of the object paradigm in the same way as the λ-calculus does for the function paradigm. However, it is still unclear how notions of encapsulation and inheritance may be modeled in π-calculus.

The notions of bisimilarity of processes can be used to verify correctness of transformations on systems. Similar to the illustration provided in [Ross 1990] to verify correctness of transformation of Prolog programs, a different application could be to model hardware circuitry using synchronization mechanisms of π-calculus. The correctness of optimization transformations on circuits could be verified by requiring that the transformations be bisimilarity preserving.

References

[Berry and Boudol 1990] G Berry and G. Boudol. The chemical abstract machine. In *Proceedings 17th Annual Symposium on Principles of Programming Languages.* 1990.

[Conery and Kibler 1985] J.S. Conery and D.F. Kibler. And parallelism and nondeterminism in logic programs. In *New Generation Computing.* 1985.

[Hehner 1984] E.C.R Hehner. Predicative programming part i. In *Communications of the ACM, 27(2).* February 1984.

[Milner and Walker 1989] J. Milner, R. Parrow and D. Walker. A calculus of mobile processes, part i. Research Report ECS-LFCS-89 85, University of Edinburgh, 1989.

[Milner 1989] R. Milner. Functions as processes. Research Report 1154, INRIA, France, 1989.

[Milner 1991] R. Milner. The polyadic π-calculus: a tutorial. Research Report, University of Edinburgh, October,1991.

[Ross 1985] B. J. Ross. On observing nondeterminism and concurrency. In *Lecture Notes in Computer Science 85.* Springer - Verlag, University of Edinburgh, 1985.

[Ross 1990] B. J. Ross. A semantic approach to prolog program analysis. In *Proceedings of UK ALP 1991, Edinburgh.* Springer - Verlag, 1990.

[Walker 1990] D. Walker. π-calculus semantics of object oriented programming languages. Research Report ECS-LFCS-90 122, University of Edinburgh, 1990.

Appendix A

Bisimilarities of representations of Prolog programs can be used to prove properties of program such as program termination, validating source-to-source transformation.

Program Termination

The π-calculus semantics can been used to reason about the termination properties of programs. As mentioned earlier finite computations eventually terminate with a *done* action, while non-terminating productive computations generate an infinite stream of *succ* actions. The looping computations or livelocks are bisimilar to the π-calculus agent $\perp$ [2] that denotes looping [Ross 1990]. The termination properties of the operators can be derived based on the termination properties of its operands. For example, the following bisimilarities can be derived using the expansion theorem, a sketch of the proof of which may be found in [Ross 1990].

$$\begin{array}{lcl} \perp \;\hat{;}\; P & \approx & \perp \\ \perp \;\triangleright\; B & \approx & \perp \\ A \;\hat{\triangleright}\; \perp & \approx & \perp \end{array}$$

Termination properties of programs can be proved by showing a terminating sequence of term rewriting based on the bisimilarities. For recursive predicate structures well-founded ordering of successive rewrites needs to be established. Consider the program in figure 9. The behavior of the program for the call $a(2)$ is looked at.

$$\begin{array}{lcll} [\ a(2)]^\circ & \approx & ([\ a_1(2)]^\circ \;\hat{;}\; [\ a_2(2)]^\circ\ )(s,d) & : \textit{by definition.} \\ & \approx & (\textit{Done} \;\hat{;}\; [\ a_2(2)]^\circ\ )(s,d) & : \textit{by resolution rule.} \\ & \approx & ([\ a_2(2)]^\circ\ )(s,d) & : \textit{by } \textbf{Seq} \textit{ bisimilarity.} \\ & \approx & (a(2))(s,d) & : \textit{by construction.} \\ & \approx & \perp & : \textit{divergence} \end{array}$$

Program Transformation

The process semantics also helps to validate source-source transformations. A standard technique in compiler optimization for imperative languages is loop unfolding. This transforms source programs with loops to ones without loops so that the transformed program executes faster than the original source program. In logic programming source programs are transformed by unfolding that replaces a goal in a clause by a prospective resolvent. Process semantics can be used to verify the validity of program transformations. A transformation of a program A to A' is valid if A *bisimilar* to A'. It has been shown by means of an example that a transformation that replaces a clause with a resolvent that has a cut in it, introduces discrepancy in the behavioral properties of the transformed program. However introduction of cut that does not affect the pruning of the search space is shown to be valid by demonstrating a bisimilarity between determinate programs before and after the introduction of cut. Determinate programs have at most one solution and are bisimilar to $(\overline{s}.\overline{d} + \overline{d})$

Partial Evaluation

[2] $\perp \overset{\text{def}}{=} (\nu x)(!\overline{x} \mid !x)$

Partial evaluation is a process of evaluating a goal at compile time with some of its arguments instantiated and deriving a new residual logic program which, when executed with the remaining input of the original program, produces the same output as the original program except that it runs more efficiently. The completeness of the partial evaluation is affected by the control strategy used by the partial evaluator. There are three stages which may use various control strategies.

1. Source Program control : the control used to execute the original source program.
2. Partial evaluation control: the control used by the partial evaluator on the source program.
3. Residual Program control : the control used to execute the residual program.

To preserve the computational completeness of the partial evaluation, it is required that the partial evaluator produce a residual program that preserves the observed behavior of the original program. This condition can be represented in π-calculus by requiring that the residual program R be bisimilar to the original program P. This condition can be maintained if only bisimilarity preserving transformations are applied during the process of partial evaluation. For example given the semantics of P to be for a left-to-right control strategy, the evaluator may not apply a right-to-left control strategy in arriving at the residual program, since the right-to-left control strategy does not preserve bisimilarity with respect to the semantics of the source program.

Given a program P and a query Q, their semantic representations are operated on by rewrite rules that preserve bisimilarities for a given control strategy. Hehner's predicative programming principle [Hehner 1984] proposes that program code and its semantic meaning are substitutive within semantic expressions. Using Hehner's predicative programming principle, the applications of bisimilarities to the program's semantic representation represent corresponding equivalence preserving transformations of the source program. Eventually a π-calculus expression E results from the rewrites. This expression is then normalized into a set of agent definitions and calls that are bisimilar to E which in turn are translated into equivalent Prolog code. This may require creation of new agents.