A Survey on π-Calculus

A Report Submitted in Partial Satisfaction of the Requirements for the Ph.D Qualifying Examination in Computer Science, Indiana University.
2014

Thejaka Amila Kanewala
School of Informatics and Computing
Indiana University, USA
thejkane@iu.edu

***Abstract*—Calculus is the mathematical study of change. Process Calculus models the changing behavior of computer processes. Usually a process refers to a running program. λ-calculus is a form of process calculus that has been heavily used in the functional programming arena. While, λ-calculus model changes in sequential computer processes, π-calculus formalizes behavior of a concurrent process. In this paper we will study general π-calculus language and variations of π-calculus that are applicable to distributed systems and programming languages.**

***Index Terms*—Pi Calculus, Process Algebra**

I. Introduction

The π-calculus provides a conceptual framework to model concurrent systems and a set of mathematical tools for expressing systems and reason about their behaviors. Before dwelve into details of π-calculus we will study the development of process calculus and how π-calculus originated from process calculus.

A. Process Algebra

A process is a series of actions or events. Algebra is a set of symbols combined according to defined rules. *Process Algebra* is a set of defined symbols along with set of rules that can be used to model computer processes. There are number of process algebras available in the literature. *Algebra of Communicating Processes* (ACP) [1], *Calculus of Communicating System* (CCS) [2] and *Communicating Sequential Processes* (CCS) [3] are some of the widely known process algebras. Though there are number of process algebras they all share following common features;

1) Compositional Modeling
Process algebras provide small number of primitive constructs to build larger parallel communicating systems. For example CCS provides six operators in total. These operators can be used to model parallel systems. Some operators are needed to compose parallel systems, some other operators are needed to select actions (choice) and rest is needed to restrict scope of actions. Compositional Modeling is mostly about syntax.
2) Operational Semantics
In *Operational Semantics* Process Algebras define the meaning of a program built using primitive constructs. Usually Process Algebras use *Structural Operational Semantics* (SOS) to derive the meaning of a program. SOS was first introduced by Plotik [4] and the basic idea behind SOS is to define the behavior of the program in terms of behavior of its sub constructs.
3) Behavioral reasoning via equivalence
Process Algebras are usually interested in capturing the notion of equivalent (or same) programs. To capture equivalent notion, process algebras construct *behavioral equivalence relations*. Identifying these equivalences are important for Process Calculus as it allows system to do various optimizations and refinements. The most common method of constructing behavioral relations is to construct *bisimulations*.

Most of the Process Algebra definitions are based on set of *Names* and set of *rules*. A Name may refer to an object, data or may even be a communication channel. Rules govern how process states should change.

B. Why Process Algebra?

Process Algebra plays a major role in program verification. During program verification we write two process algebraic specifications. First specification models the actual system implementation and second specification describes the desired "high-level" system behavior. One may prove the correctness of the system implementation by proving that behavior of first (implemented) specification is "as same as" the behavior of the second (high-level system behavior) specification.

Checking whether pair of specifications is the "same" can be done in two ways. First method is to compare two specifications in syntax oriented manner. Second method is to compare specifications in semantics oriented manner. In syntax-oriented case, we perform transformations on the specification syntax according to set of defined rules in order to derive one syntax from another. In semantic comparison we build a behavioral relation based on operational semantics to check whether 2 specifications behave in the same way.

Process algebra theory is also used in programming language development; mainly during type checking and during optimizations.

C. π-Calculus

π-calculus is a recent addition to the process algebra family. Main difference between π-calculus and other process algebra is the ability to communicate channel names. Because of this feature, π-calculus is specifically useful in describing concurrent computations where network configurations change during program execution.

The basic computational step in π-calculus is the transfer of a communication link between two processes. When sender sends the communication link the recipient can use the communication link for its subsequent communication. Our detail discussion about π-calculus starts with the example described below.

D. An Example

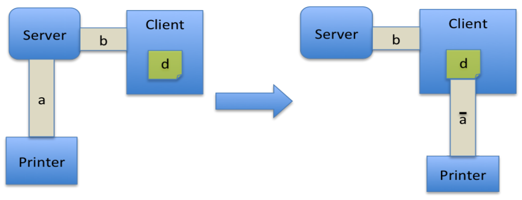

Fig. 1: π-calculus example

There is a printer server (S), which handles all printing requests from other clients (C as client). The client has document "d" to be printed. In a real implementation client will first authenticate and authorize to use the printer. Then client sends the document to the server and server redirect document data to the printer.

π-calculus models above scenario in terms of the behavior of client and server. Before presenting exact formula lets first get us familiar with some π-calculus terms.

- $\bar{b}a$ - Sending name a through channel name b
- $b(a)$ - Receiving a name in channel name b and the value is represented using name "a"
- S - Behavior of the printer server after authenticating and authorizing the channel
- P - Behavior of the client after sending the document to printer

Above interaction is formulated using π-calculus as follows;

$$\bar{b}a.S|b(c).\bar{c}d.P \longrightarrow S|\bar{a}d.P \tag{1}$$

In π-calculus each parallel executing component is separated by "$|$". Above transition is interpreted as follows; The server sends name "a" through channel name b ($\bar{b}a$) and afterwards server goes to its normal operation (S) (waiting for new client connections). The client receives "c" through channel "b" ($b(c)$) and sends the document to printer through received channel ($\bar{c}d$). Afterwards client goes to its normal operation (depicted by "P"). After exchanging channel "a" between server and client will perform as $S|\bar{a}d.P$.

As per above example the channel name (a) is treated in the same way as data object (d). Infact this is the reason why π-calculus is different from other process algebras. Intuitively we can think of transferring "a" represent granting access rights to client. Then client uses those access rights to access the printer. Another interpretation is that printer is "moved" to the client (if channel "a" is the only way to communicate with printer). Based on this interpretation π-calculus is also called a calculus of *mobile processes*.

Most process algebras have a way to declare a communication link local to a set of processes. As an example processes P and Q may share channel "a" to exchange some data. In π-calculus such situations are called *restrictions* (restricting channel "a" only to P and Q processes). π-calculus models restricting behaviours as $(\nu a)(P|Q)$. In above example suppose R represent the printer process. Then initially "a" is a channel name private to server (S) and printer (R). Therefore using π-calcuclus we can write $(\nu a)(\bar{b}a.S|R)$.

Since π-calculus treats communication links as first class entities we can transfer links to other processes. Therefore we can transfer private link "a" to client (P) process and this behaviour is modeled as given in 2;

$$(\nu a)(\bar{b}a.S|R)|b(c).\bar{c}d.P \xrightarrow{\tau} (\nu a)(S|R|\bar{a}d.P) \tag{2}$$

E. About This Paper

Though the central idea of π-calculus, is developed based on process algebra with link passing, there are number of variations and descendants of π-calculus. In this paper we are more focused on general π-calculus definitions; but whenever applicable we will discuss variations of π-calculus and their applicability. Through this paper we do not wish to introduce any new concept or theorem. We discuss and summarize standard π-calculus with several worked examples. Rest of the paper is organized as follows; In Section II we will discuss the syntax of π-calculus and structural congruence; in Section III we will discuss the operational semantics. Behavioral equivalence and bisimulations are discussed in Section IV. Few π-calculus variations are summarised in Section V. In VI and VII we will discuss how π-calculus is related to distributed computing and programming languages.

The paper contents are based on following references; the process algebra definitions are mostly based on reference [5], core π-calculus definitions are based on references [5], [6], [7]. The π-calculus variations are based on references [8], [9].

II. π-CALCULUS SYNTAX

The basic entities in π-calculus are called *Names*. Names represent communication channels, variables and data values. π-calculus also defines set of agent identifiers, each with a fixed non-negative arity. Processes evolve by performing actions. The capabilities for an action are expressed through *prefixes*. The syntax of π-calculus is given in Table 1.

Prefixes	α	::=	$\bar{a}x$	Output
			$a(x)$	Input
			τ	Silent
Agents	P, Q	::=	$\mathbf{0}$	Nil
			$\alpha.P$	Prefix
			$P + Q$	Sum
			$P\|Q$	Parallel
			if x=y *then P*	Match
			if x≠y *then P*	Mismatch
			$(\nu x)P$	Restriction
			$!P$	Replication
			$A(y_1, ..., y_n)$	Identifier

Definitions $A(x_1, ..., x_n) \overset{\text{def}}{=} P \; where \; i \neq j \Rightarrow x_i \neq x_j$

Table 1: The syntax of π-calculus

As per Table 1, an agent can be in one of the following forms;

1) **0** - The empty agent cannot perform any action.
2) α - Prefix state
   a) Output prefix $\bar{a}x.P$: The name x is sent over channel a and afterwards agent continues to act as P.
   b) Input prefix $a(x).P$: A name is received through channel a and x is a placeholder for the received name. After input is received agent will continue processing as P.
   c) Silent prefix $\tau.P$: An agent that can evolve to P without interacting with other agents.

   Many references use α, β to represent prefixes. We will also follow the same approach. Further we say that input prefix and output prefix has a *subject* and an *object*. For example in output prefix $\bar{a}x.P$, "a" is the subject and "x" is the object. In input prefix $a(x).P$, "a" is the subject and "x" is the object.
3) *Sum* $P + Q$: An agent that can act on either P or Q.
4) *Parallel Composition* - $P|Q$: Represents combined behavior of P and Q executing in parallel. Elements separated by "|" are called *components*. Components P and Q can act independently and may also communicate if P has a send and Q has a receive on the same channel.
5) *Match* - $if \; x = y \; then \; P$: The agent will behave as P if x and y are the same name, otherwise it does not.
6) *Mismatch* - $if \; x \neq y \; then \; P$: If x and y are "not" the same name then agent will behave as P, otherwise agent will not behave as P.
7) *Restriction* - $(\nu x)P$: The name x is local to P. The agent will behave as P but agent will not be able to use x to communicate between components in P and components outside $(\nu x)P$ expression; x can be only used to communicate between components inside P.
8) *Replication* - $!P$: Infinite composition $P|P|\cdots$; equivalently, a process satisfying the equation $!P = P|!P$. Replication enables us to define infinite behaviors.
9) *Identifier* - $A(y_1, \ldots, y_n)$ where n is the arity of A : Every identifier correspond to a *Definition* $A(x_1, \ldots, x_n) \overset{def}{=} P$ where each x_i is pairwise distinct. We can think that $A(y_1, \ldots, y_n)$ behave as P by replacing each x_i with y_i. Further definition is similar to function decleration with $x_1, \ldots, x_n$ as formal parameters and the identifier $A(y_1, \ldots, y_n)$ similar to an real invocation with actual parameters $y_1, \ldots, y_n$.

In following we will go through several π-calculus examples to get a better understanding about each syntactic constructs explained above.

Example 1. $x(z).\bar{y}z.\mathbf{0}$ *- Process receives a name via channel x and received name is identified by z. Then channel y is used to send received name (z). After sending z via y process becomes inactive.*

Example 2. $x(z).\bar{z}y.\mathbf{0}$ *- Process receives a name via x and send y via the name received (z) and become inactive.*

Example 3. $x(z).if \; z = y \; then \; \bar{z}w.\mathbf{0}$ *- A name is received via x and if received name is same as y then send w via received name. Otherwise process will not do anything further.*

Example 4. $x(z).y(w).if \; z = w \; then \; \bar{v}u.\mathbf{0}$ *- Process receives 2 names from channels x and y, if those 2 names are same process send u via v, otherwise process will come to an inactive state.*

Example 5. $x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0}$ *- Process either recieves a name via x and send y via received name and become inactive "OR" send v via w and become inactive.*

Example 6. $(x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0})|\bar{x}u.\mathbf{0}$ *- The process has 2 components. Those components can behave independently from one another. Therefore the process has 4 capabilities. The first component may receive a name via x and use that received name to send y and become inactive "OR" send v via w and become inactive. The second component send u via x and become inactive. Further since x is shared between first component and second component they can silently evolve.*

Example 7. $(\nu x)((x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0})|\bar{x}u.\mathbf{0})$ *- Unlike previous example, this statement has one 2 capabilities. In this example x is a restricted channel. One capability is to send v via w and other capability is to evolve silently due to the interaction between 2 components through x.*

Example 8. $!x(z).!\bar{y}z.\mathbf{0}$ *- The process can receive names via x repeatedly, and can repeatedly send received names via y.*

Next we will discuss how we can establish, two π-calculus terms are syntactically equivalent (also called *structural equivalence*). Before going into details of structural equivalence we need to discuss several related definitions. In next subsections we will discuss two related topics to *structural equivalence*; namely *Bindings* and *Substitution*.

A. Binding

From the constructs described in Table 1, the input Prefix and Restriction constructs are different from others in a special way; i.e. both input Prefix and Restriction *bind* names.

The input Prefix $a(x).P$ said to *bind* x in P, and occurrences of x in P are then called *bound*. In contrast the output Prefix $\bar{a}x.P$ does not bind x. In summary the object is bound in input Prefixes and object is *free* in output prefixes. The silent Prefix τ does not have an object or subject; therefore it does not have the notion of binding. Further the Restriction operator $(\nu x)P$ binds x in P. Same kind of bindings can be seen in other process algebras ($\backslash x$ in CCS and δ_x in ACP) as well. But in other process algebras we cannot pass channel names between agents. But in π-calculus we can transfer channel names between agents.

In following we give the formal definition of a Binding.

Definition 1. ***(Binding of a variable)***
In each of $x(z).P$ and $(\nu z)P$, the displayed occurrence of z is ***binding*** *with* ***scope*** *P. An occurrence of a name in a process is* ***bound*** *if it is, or it lies within the scope of, a binding occurrence of the name. An occurrence of a name in a process is* ***free*** *if it is not bound.*

We write $fn(P)$ for the set of names that have a free occurrence in P and $bn(P)$ to denote the set of bound names in P.

Example 9. $fn((\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0}|\bar{x}u.\mathbf{0}) = \{z, y, w, v, x, u\}$ *and* $bn((\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0}|\bar{x}u.\mathbf{0}) = \{\}$.

Example 10. $fn((\nu x)((x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0})|(\nu u)\bar{x}u.\mathbf{0})) = \{y, w, v\}$ *and*
$bn((\nu x)((x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0})|(\nu u)\bar{x}u.\mathbf{0})) = \{x, z, u\}$.

B. Substitution

A *substitution* is a function from Names to Names. We write x/y for the substitution that maps y to x and is identity for all other names. In general we use $\{x_1, \ldots, x_n/y_1, \ldots, y_n\}$ to denote substitution where each y_i are pairwise distinct, for a function that maps each y_i to x_i.

As a notation we use σ to range over substitutions. The agent $P\sigma$ is P where all free names x are replaced by $\sigma(x)$. But the replacement must however be done in such a way that unintended captures of names by binders is avoided. For example if we try to execute $a(x).\bar{x}y\{x/y\}$ we would get $a(x).\bar{x}x$ which gives a complete different semantics from the original expression. A correct way to execute substitution would be to first substitute x with a different name other than x and y and then perform substitution on y. i.e. $a(x).\bar{x}y\{z/x\}\{x/y\} = a(z).\bar{z}x$. This behavior is quite similar to the λ-calculus substitutions. In λ-calculus we use α-conversion to avoid unintended capture replacement. In π-calculus we will use a similar convertible rule to avoid unintended substitutions.

Definition 2. ***(α-convertible)***

1) *If the name w does not occur in the process P, then P{w/z} is the process obtained by replacing each free occurrence of z in P by w.*
2) *A change of bound names in a process P is the replacement of a subterm $x(z).Q$ of P by $x(w).Q\{w/z\}$, or the replacement of a subterm $(\nu z)Q$ of P by $(\nu w)(Q\{w/z\}$, where in each case w does not occur in Q.*

Example 11. $(y(w).\bar{w}x.\mathbf{0})\{z/x\} = y(w).\bar{w}z.\mathbf{0}$

Example 12. $(!(\nu z)\bar{x}z.\mathbf{0}|y(w).\mathbf{0})\{v, v/x, y\}$
$=!(\nu z)\bar{v}z.\mathbf{0}|v(w).\mathbf{0}$

More formally we define application of substitution (σ) as follows;

Definition 3. ***(Application of substitution)*** *The process $P\sigma$ obtained by applying σ to P is defined as follows;*

$$\begin{aligned}
\mathbf{0}\sigma &\overset{def}{=} \mathbf{0} \\
(\alpha.P)\sigma &\overset{def}{=} \alpha\sigma.P\sigma \\
(P + P')\sigma &\overset{def}{=} P\sigma + P'\sigma \\
(P|P')\sigma &\overset{def}{=} P\sigma|P'\sigma \\
((\nu z)P)\sigma &\overset{def}{=} (\nu z)P\sigma \\
(!P)\sigma &\overset{def}{=} !P\sigma
\end{aligned}$$

C. Structural Congruence

The agents $a(x).\bar{b}x$ and $a(y).\bar{b}y$ are syntactically different. But both depict the same behaviour; i.e. an agent that receives a name along channel a and sends it through channel b. Further the order in parallel composition ($P|Q$) is not important. We can either have $P|Q$ or $Q|P$. But still statements $P|Q$ and $Q|P$ are syntactically different.

To identify agents, which intuitively represent the same meaning, we use a relation called *Congruence*. *Structural Congruence* is a form of congruence relation that is defined based on π-calculus syntax. Before delving into structural congruence we will first give a definition to general congruence relation. To define congruence formally we need two helper definitions.

Definition 4. ***(Degenerate and Non-degenerate)*** *An occurrence of* **0** *in a process is degenerate if it is the left or right term in a sum $M_1 + M_2$, and non-degenerate otherwise.*

Definition 5. ***(Context)*** *A context is obtained when the hole [.] replaces a non-degenerate occurrence of* **0** *in a process-term given by the grammar in Table 1.*

To clarify the definition we will look into an example context.

Example 13. *An example context -* $C_0 = (\nu z)([.]!z(w).\bar{w}a.\mathbf{0})$

If C is a context and P a process, we write C[P] for the process obtained by replacing the [.] in C by P. The replacement is purely literal. An example is as follows;

$$C_0[!\bar{z}b.\mathbf{0}] = (\nu z)(!\bar{z}b.\mathbf{0}|!z(w).\bar{w}a.\mathbf{0})$$

Now we can define the congruence formally. In following we give congruence definition.

Definition 6. ***(Congruence)*** *An equivalence relation S on processes is a congruence if $(P, Q) \in S$ implies $(C[P], C[Q]) \in S$ for every context C.*

Structural Congruence is a congruence relation to identify agents, which intuitively represent the same meaning (syntactically). The structural congruence is purely based on the syntax and *structure* and does not deal with the semantics. There are several different versions of structural congruence. Our structural congruence definition is based on reference [6]. The structural congruence definition is given in Table 2.

Definition 7. ***Structural Congruence*** *The structural congruence $\equiv$ is defined as the smallest congruence satisfying the following laws;*

1) *If P and Q are variants of α-conversion then $P \equiv Q$.*
2) *The Abelian monoid laws for Parallel*
   a) *Commutativity : $P|Q \equiv Q|P$*
   b) *Associativity : $(P|Q)|R \equiv P|(Q|R)$*
   c) ***0** as unit : $P|\mathbf{0} \equiv P$*
3) *The Abelian monoid laws for Sum*
   a) *Commutativity : $P + Q \equiv Q + P$*
   b) *Associativity : $(P + Q) + R \equiv P + (Q + R)$*
   c) ***0** as unit : $P + \mathbf{0} \equiv P$*
4) *The unfolding law $A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}\ if\ A(\tilde{x}) \overset{def}{=} P$*
5) *The scope extension laws*
   a) $(\nu x)\mathbf{0} \equiv \mathbf{0}$
   b) $(\nu x)(P|Q) \equiv P|(\nu x)Q\ \ if\ x \notin fn(P)$
   c) $(\nu x)(P + Q) \equiv P + (\nu x)Q\ \ if x \notin fn(P)$
   d) $(\nu x)\ if\ \ u\ = \ v\ \ then\ \ P\ \equiv\ \ if\ u\ = v\ then\ (\nu x)P\ \ if x \neq u\ and\ x \neq v$
   e) $(\nu x)\ if\ \ u\ \neq\ v\ \ then\ \ P\ \equiv\ \ if\ u\ \neq v\ then\ (\nu x)P\ \ if x \neq u\ and\ x \neq v$
   f) $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
6) *The equational reasoning laws*
   a) *Refl $P \equiv P$*
   b) *Symm $P \equiv Q \rightarrow Q \equiv P$*
   c) *Trans $P \equiv Q\ and\ Q \equiv R \rightarrow P \equiv R$*

Table 2: The definition of *Structural Congruence*

The α-conversion identifies agents like $a(x).\bar{b}x$ and $a(y).\bar{b}y$ as the same. *Abelian Monoid* laws for Sum and Parallel basically says that Parallel and Sum are unordered operations. The unfolding just states that Identifier is same as it definition with appropriate parameter instantiation. The scope extension rules say that if private channel is used in only within a single component then we can move private channel to that component. For example in parallel composition if all occurrences are in one of the components then it does not matter if the Restriction covers only that component or the whole composition. In following we will look into an example where structural congruence is used to prove 2 π-calculus statements are syntactically equivalent.

Example 14. *We prove $P \equiv (\nu x)P$*
$P \equiv P|\mathbf{0}$ (Using 2-c)
$P|\mathbf{0} \equiv P|(\nu x)\mathbf{0}$ (Using 5-a)
$P|(\nu x)\mathbf{0} \equiv (\nu x)(P|\mathbf{0})$ ($x \notin fn(\mathbf{0})$ and 5-b)
$(\nu x)(P|\mathbf{0}) \equiv (\nu x)P$ (Again 2-c)
Therefore $P \equiv (\nu x)P$.

Reductions and Structural Congruence goes hand in hand. Therefore we will look into examples after discussing reductions. In next subsection we start the discussion of reductions.

D. Reductions

Consider following process;

$$a(x).\bar{c}x|\bar{a}b$$

Above process has 2 components. The first component accepts an input from channel a and second component sends a name through channel a. Therefore we can simulate above process as first component replacing all its x names with b and second component sending b and afterwards becoming Nil. This behaviour is similar to Silent Transition (τ transition). So the transition would look like as follows;

$$a(x).\bar{c}x|\bar{a}b \xrightarrow{\tau} \bar{c}b|\mathbf{0}$$

Doing reduction based on syntax would greatly help to check whether 2 processes are the same. Therefore π-calculus introduces set of axioms to perform reductions based on syntax. These axioms give the same effect as silent action.

The assertion $P \longrightarrow P'$ express that process P can evolve to process P' as a result of intra-action, that is an action *within* P. Reduction is defined by a family of inference rules.

Definition 8. ***(Reduction Semantics)*** *The reduction relation, $\longrightarrow$, is defined by the rules in Table 3.*

$$\frac{}{(\bar{x}y.P_1 + M_1)|(x(z).P_2 + M_2) \longrightarrow P_1|P_2\{y/z\}} \quad (1)$$

$$\frac{}{\tau.P + M \longrightarrow P} \quad (2)$$

$$\frac{P_1 \longrightarrow P_1'}{P_1|P_2 \longrightarrow P_1'|P_2} \quad (3) \qquad \frac{P \longrightarrow P'}{(\nu z)P \longrightarrow (\nu z)P'} \quad (4)$$

$$\frac{P_1 \equiv P_2 \longrightarrow P_2' \equiv P_1'}{P_1 \longrightarrow P_1'} \quad (5)$$

Table 3: Reduction inference rules.

Informally processes P and Q stand in the reduction relation just if the assertion $P \longrightarrow Q$ can be inferred via the rules. The assertion expresses that P can evolve to Q. Lets have a closer look at above rules.

Consider the process denoted by left hand side of the rule (1) (in denominator). This process has 2 components. The first component can send name y through channel x or act as M_1. The second component takes a name through channel x as an input or act as M_2. The rule says that process has a reduction due to the interaction between its components via channel x and as a result y is passed from the first component to second component and is substituted for the placeholder z in P_2. The two prefixes are consumed and other behaviours, namely M_1 and M_2 are rendered void.

The rule (2) says the process can evolve due to a silent action. In this case also the additional capability (i.e. M) is rendered void. The rules (3) and (4) are similar. (3) says if the component P_1 has a reduction then process $P_1|P_2$ has a reduction. Similar behaviour is applied for restrictions.

In following we will look into several examples where we use reductions and structural congruence in combination to reduce processes.

Example 15. *Suppose* $P \overset{def}{=} (\nu x)(x(z).\bar{z}y.\mathbf{0}|(\bar{x}a.\mathbf{0}|\bar{x}b.\mathbf{0}))$
We need to reduce P.
Using associativity rule (Table 2, 3-b) we have;
$P \equiv (\nu x)((x(z).\bar{z}y.\mathbf{0}|\bar{x}a.\mathbf{0})|\bar{x}b.\mathbf{0})$
Using commutavity rule (Table 2, 3-a) we have;
$P \equiv (\nu x)((\bar{x}a.\mathbf{0}|x(z).\bar{z}y.\mathbf{0})|\bar{x}b.\mathbf{0})$
Using **0** *introduction (Table 2, 3-c) we have;*
$P \equiv (\nu x)((\bar{x}a.\mathbf{0}+\mathbf{0}|x(z).\bar{z}y.\mathbf{0}+\mathbf{0})|\bar{x}b.\mathbf{0})$
Now, let $P_1 \overset{def}{=} (\bar{x}a.\mathbf{0}+\mathbf{0})|((x(z).\bar{z}y.\mathbf{0}+\mathbf{0})$. *Then we have* $P \equiv (\nu x)(P_1|\bar{x}b.\mathbf{0})$
Consider $P_1 \overset{def}{=} (\bar{x}a.\mathbf{0}+\mathbf{0})|((x(z).\bar{z}y.\mathbf{0}+\mathbf{0})$;
Using Table 3, rule (1), we have $P_1 \longrightarrow \mathbf{0}|(\bar{z}y.\mathbf{0})\{a/z\}$
Therefore $P_1 \longrightarrow \mathbf{0}|\bar{a}y.\mathbf{0}$
Further, using Table 3 rule (3);

$$\frac{P_1 \longrightarrow \mathbf{0}|\bar{a}y.\mathbf{0}}{P_1|\bar{x}b.\mathbf{0} \longrightarrow (\mathbf{0}|\bar{a}y.\mathbf{0})|\bar{x}b.\mathbf{0}}$$

Then using Table 3, rule (4) we have following;

$$\frac{P_1|\bar{x}b.\mathbf{0} \longrightarrow (\mathbf{0}|\bar{a}y.\mathbf{0})|\bar{x}b.\mathbf{0}}{(\nu x)(P_1|\bar{x}b.\mathbf{0}) \longrightarrow (\nu x)((\mathbf{0}|\bar{a}y.\mathbf{0})|\bar{x}b.\mathbf{0})}$$

Further we know (Table 2, rule 3-c) $((\mathbf{0}|\bar{a}y.\mathbf{0})|\bar{x}b.\mathbf{0}) \equiv (\bar{a}y.\mathbf{0}|\bar{x}b.\mathbf{0})$
Then using Table 3, rule (5) we get below result; $(\nu x)(P_1|\bar{x}b.\mathbf{0}) \longrightarrow (\nu x)(\bar{a}y.\mathbf{0}|\bar{x}b.\mathbf{0})$
Finally we have $P \longrightarrow (\nu x)(\bar{a}y.\mathbf{0}|\bar{x}b.\mathbf{0})$.

Example 16. *Let* $Q \overset{def}{=} (\nu x)((x(y).x(z).\bar{y}z.\mathbf{0}|$
$x(w).x(v).\bar{v}w.\mathbf{0}|\bar{x}a.\bar{x}b.\mathbf{0})$.
We need to reduce Q.
Using commutavity rule (Table 2, 3-a) we have;
$Q \equiv (\nu x)(\bar{x}a.\bar{x}b.\mathbf{0}|(x(y).x(z).\bar{y}z.\mathbf{0}|$
$x(w).x(v).\bar{v}w.\mathbf{0}))$
Then using associativity rule (Table 2, 3-b) we have;
$Q \equiv (\nu x)((\bar{x}a.\bar{x}b.\mathbf{0}|x(y).x(z).\bar{y}z.\mathbf{0})|$
$x(w).x(v).\bar{v}w.\mathbf{0})$
With **0** *introduction (Table 2, 3-c)*
$Q \equiv (\nu x)(((\bar{x}a.\bar{x}b.\mathbf{0}+\mathbf{0})|(x(y).x(z).\bar{y}z.\mathbf{0}+\mathbf{0}))|$
$x(w).x(v).\bar{v}w.\mathbf{0})$
Let $P \overset{def}{=} (\bar{x}a.\bar{x}b.\mathbf{0}+\mathbf{0})|(x(y).x(z).\bar{y}z.\mathbf{0}+\mathbf{0})$. *Then* $Q \equiv (\nu x)(P|x(w).x(v).\bar{v}w.\mathbf{0})$.
Then we can apply Table 3, rule (1) as follows;

$$\frac{}{(\bar{x}a.\bar{x}b.\mathbf{0}+\mathbf{0})|(x(y).x(z).\bar{y}z.\mathbf{0}+\mathbf{0}) \longrightarrow \bar{x}b.\mathbf{0}|x(z).\bar{y}z.\mathbf{0}\{a/y\}}$$

Which results in following;

$$\frac{}{(\bar{x}a.\bar{x}b.\mathbf{0}+\mathbf{0})|(x(y).x(z).\bar{y}z.\mathbf{0}+\mathbf{0}) \longrightarrow \bar{x}b.\mathbf{0}|x(z).\bar{a}z.\mathbf{0}}$$

Further using **0** *introduction we have following;*
$\bar{x}b.\mathbf{0}|x(z).\bar{a}z.\mathbf{0} \equiv \bar{x}b.\mathbf{0}+\mathbf{0}|x(z).\bar{a}z.\mathbf{0}+\mathbf{0}$
Now we apply Table 3 rule (1) to right hand side of above equivalence as follows;

$$\frac{}{(\bar{x}b.\mathbf{0}+\mathbf{0})|(x(z).\bar{a}z.\mathbf{0}+\mathbf{0}) \longrightarrow \mathbf{0}|\bar{a}b.\mathbf{0}}$$

Then from Table 3 rule (5) we get following result;

$$\frac{\bar{x}b.\mathbf{0}|x(z).\bar{a}z.\mathbf{0} \equiv \bar{x}b.\mathbf{0}+\mathbf{0}|x(z).\bar{a}z.\mathbf{0}+\mathbf{0} \longrightarrow \mathbf{0}|\bar{a}b.\mathbf{0} \equiv \bar{a}b.\mathbf{0}}{\bar{x}b.\mathbf{0}|x(z).\bar{a}z.\mathbf{0} \longrightarrow \bar{a}b.\mathbf{0}}$$

From last 4 steps we derived $P \longrightarrow \bar{a}b.\mathbf{0}$. *Now applying Table 3, rule (3)*

$$\frac{P \longrightarrow \bar{a}b.\mathbf{0}}{P|x(w).x(v).\bar{v}w.\mathbf{0} \longrightarrow \bar{a}b.\mathbf{0}|x(w).x(v).\bar{v}w.\mathbf{0}}$$

Applying Table 3, rule (4)

$$\frac{P|x(w).x(v).\bar{v}w.\mathbf{0} \longrightarrow \bar{a}b.\mathbf{0}|x(w).x(v).\bar{v}w.\mathbf{0}}{(\nu x)(P|x(w).x(v).\bar{v}w.\mathbf{0}) \longrightarrow (\nu x)(\bar{a}b.\mathbf{0}|x(w).x(v).\bar{v}w.\mathbf{0})}$$

Since $Q \equiv (\nu x)(P|x(w).x(v).\bar{v}w.\mathbf{0})$ *we obtain final result by applying Table 3, rule (5), which yields following;*
$Q \longrightarrow (\nu x)(\bar{a}b.\mathbf{0}|x(w).x(v).\bar{v}w.\mathbf{0})$.

So far we have been looking at syntactic behaviour of π-calculus. In next section we will look at how we can give meaning to syntactic constructs we defined in Table 1.

III. Operational Semantics

Activity within a system can be described using reduction relation. But reductions do not explain how a system can interact with its environment (input/output). In order to understand the behaviour of the system we analyse the behaviour of actions performed by each sub-part. This is the same SOS approach that is used to define the operational semantics. In this section we discuss about Operational Semantics.

In the literature of π-calculus we find several operational semantic definitions for π-calculus. In this paper we discuss widely used methodology to encode operational semantics to π-calculus. This standard definition is based on a *labeled transition system*(LTS).

In LTS the transitions would look like $P \xrightarrow{\theta} Q$ for some set of actions ranged over θ. It is important to identify possible actions within the system (in other words possible values for θ). For an agent $\alpha.P$ there will be a transition labelled α leading to P. Further Restriction operator will not permit an action with the restricted name as subject, i.e. $(\nu x)\bar{x}u$ has no transition and therefore it has the same effect as **0** (Nil). But the behaviour is different when restricted name appear as an object. When the restricted name is an object we should have an action. But intuitively the action does not belong to any of the actions we defined under Prefixes (Input, Output or Silent). We further analyse the situation with an example.

Consider process $(\nu u)\bar{a}u.P$. In this example the bound variable u is an object. To convince ourselves that $(\nu u)\bar{a}u.P$ has an action we will consider another agent which has 2 components and one of the component is $(\nu u)\bar{a}u.P$. The process we are considering is $a(x).Q|(\nu u)\bar{a}u.P$. Since $u \notin fn(Q)$ by structural congruence we can have $a(x).Q|(\nu u)\bar{a}u.P \equiv (\nu u)(a(x).Q|\bar{a}u.P)$. There is an interaction between components in expression, $(\nu u)(a(x).Q|\bar{a}u.P)$. Therefore clearly $(\nu u)\bar{a}u.P$ cannot behave as Nil (**0**) process. Then can $(\nu u)\bar{a}u.P$ act $\bar{a}u$? We prove that is not the case; using example below;

Consider a component $(a(x).if\, x = u\; then\; Q)$. We parallel compose this component with $\bar{a}u$ and $(\nu u)\bar{a}u$ to see whether we derive the same output.

Using reduction rules we can prove $(a(x).\; if\; x = u\; then\; Q)|\bar{a}u \longrightarrow if\; u = u\; then\; Q$ which continues as Q......(a) Now consider $(a(x).\; if\; x = u\; then\; Q)|(\nu u)\bar{a}u$. Then $(a(x).\, if\, x = u\, then\, Q)|(\nu u)\bar{a}u \equiv (\nu v)((a(x).\, if\, x = u\, then\; Q)|\bar{a}v) \longrightarrow (\nu v)if\; v = u\; then\; Q$.......(b).

As per (a) and (b) the results we got by parallel composing $(a(x).\; if\; x = u\; then\; Q)$ with $\bar{a}u$ and parallel composing $(a(x).if\, x = u\, then\, Q)$ with $(\nu u)\bar{a}u$ are different. Therefore we conclude that $(\nu u)\bar{a}u$ does *not* behave same as $\bar{a}u$.

All of above discussion convince that we need to have a new action type for restrictions that bind objects. Therefore $(\nu u)\bar{a}u$ is given action called *bound output* (which is different from the actions defined for Prefix) and written as $\bar{a}\nu u$. More formally the bound output is a combination of an output and restriction and can be defined as $\bar{a}\nu x.P = (\nu x)\bar{a}x.P$. Intuitively the bound output says we are sending a fresh name through the channel.

In summary, θ ranged over following actions;

1) The internal action - τ
2) Sending name x via a - $\bar{a}x$.
3) Receiving x via a - $a(x)$
4) Sending a *fresh* name via x - $\bar{a}\nu x$

Definition 9. ***Action*** *The actions (θ) is given by;*

$$\theta ::= \bar{a}y|a(y)|\bar{a}\nu y|\tau$$

Definition 10. ***(Operational Semantics)*** *The operational semantics (denoted $\xrightarrow{\theta}$) for π-calculus ranged over θ is defined in Table 4.*

STRUCT

$$\frac{P' \equiv P, P \xrightarrow{\theta} Q, Q \equiv Q'}{P' \xrightarrow{\theta} Q'}$$

OUT

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$$

INP

$$\frac{}{x(y).P \xrightarrow{x(z)} P\{z/y\}}$$

TAU

$$\frac{}{\tau.P \xrightarrow{\tau} P}$$

SUM

$$\frac{P \xrightarrow{\theta} P'}{P + Q \xrightarrow{\theta} P'}$$

MATCH

$$\frac{P \xrightarrow{\theta} P'}{if\; x = x\; then\; P \xrightarrow{\theta} P'}$$

MISMATCH

$$\frac{P \xrightarrow{\theta} P', x \neq y}{if\; x \neq y\; then\; P \xrightarrow{\theta} P'}$$

PAR

$$\frac{P \xrightarrow{\theta} P', bn(\alpha) \cap fn(Q) = \emptyset}{P|Q \xrightarrow{\theta} P'|Q}$$

COM

$$\frac{P \xrightarrow{a(x)} P', Q \xrightarrow{\bar{a}u} Q'}{P|Q \xrightarrow{\tau} P'\{u/x\}|Q'}$$

RES

$$\frac{P \xrightarrow{\theta} P', x \notin names(\alpha)}{(\nu x)P \xrightarrow{\theta} (\nu x)P'}$$

OPEN

$$\frac{P \xrightarrow{\bar{a}x} P', a \neq x}{(\nu x)P \xrightarrow{\bar{a}\nu x} P'}$$

CLOSE

$$\frac{P \xrightarrow{\bar{a}\nu z} P', Q \xrightarrow{a(z)} Q', z \notin fn(Q)}{P|Q \xrightarrow{\tau} (\nu z)(P'|Q')}$$

Table 4: The standard operational semantics for π-calculus.

The $names\;(\alpha)$ represent all the names present in α (both bound and free). To keep the discussion concise we skip the inference rules for Replication. The STRUCT rule basically says that if 2 terms are structurally congruent then they have the same operational semantics. COM rule is used in many situations where local communication is possible. OPEN is the rule that generates bound outputs.

In following we go through few examples.

Example 17. *Consider process $x(z).\bar{z}y.\boldsymbol{0}$ with an arbitary input name a.*
Using INP we observe following result.

$$\frac{}{x(z).\bar{z}y.\boldsymbol{0} \xrightarrow{x(a)} \bar{a}y.\boldsymbol{0}}$$

Further using OUT $\bar{a}y.\boldsymbol{0} \xrightarrow{\bar{a}y} \boldsymbol{0}$
Therefore $x(z).\bar{z}y.\boldsymbol{0} \longrightarrow \boldsymbol{0}$.

Example 18. *Consider $x(z).if\; z = y\, then\; \bar{z}w.\boldsymbol{0}$ with input y via x. We first apply INP rule.*
$x(z).if\; z = y\; then\; \bar{z}w.\boldsymbol{0} \xrightarrow{x(y)} if\; y = y\; then\; \bar{y}w.\boldsymbol{0}$
Now we apply MAT rule.

$$\frac{\bar{y}w.\boldsymbol{0} \xrightarrow{\bar{y}w} \boldsymbol{0}}{if\; y = y\; then\; \bar{y}w.\boldsymbol{0} \xrightarrow{\bar{y}w} \boldsymbol{0}}$$

Example 19. *Consider $x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0}$ with input a via x. Applying INP to $x(z).\bar{z}y.\mathbf{0}$ we get following;*
$x(z).\bar{z}y.\mathbf{0} \xrightarrow{x(a)} \bar{a}y.\mathbf{0}$
Further $\bar{a}y.\mathbf{0} \xrightarrow{\bar{a}y} \mathbf{0}$ Now we apply SUM rule with above outcome;

$$\frac{x(z).\bar{z}y.\mathbf{0} \xrightarrow{x(a)} \mathbf{0}}{x(z).\bar{z}y.\mathbf{0} + \bar{w}v.\mathbf{0} \xrightarrow{x(a)} \mathbf{0}}$$

Example 20. *Consider following process;*
$P \overset{def}{=} ((\nu s)\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0})|x(w).(w(v).w(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0})$
To check the behaviour of above process we first consider $\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0}$ part. Then using OUT we can have a derivation as follows;

$$\frac{}{\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0} \xrightarrow{xs} \bar{s}a.\bar{s}b.\mathbf{0}}$$

Applying OPEN;

$$\frac{\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0} \xrightarrow{xs} \bar{s}a.\bar{s}b.\mathbf{0}}{(\nu s)\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0} \xrightarrow{\bar{x}\nu s} \bar{s}a.\bar{s}b.\mathbf{0}}$$

Now we focus on other part of P; i.e. $x(w).(w(v).w(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0})$. We apply INP rule on this component with input $x(s)$.
$x(w).(w(v).w(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0}) \xrightarrow{x(s)} s(v).s(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0}$

Now we use $(\nu s)\bar{x}s.\bar{s}a, \bar{s}b.\mathbf{0} \xrightarrow{\bar{x}\nu s} \bar{s}a.\bar{s}b.\mathbf{0}$ and $x(w).(w(v).w(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0}) \xrightarrow{x(s)} s(v).s(u).\bar{v}u.\mathbf{0}$ to apply CLOSE rule. CLOSE rule will yield us following behaviour;
$P \xrightarrow{\tau} (\nu s)(\bar{s}a.\bar{s}b.\mathbf{0}|(s(v).s(u).\bar{v}u.\mathbf{0}|z(t).\mathbf{0}))$

Example 21. *We will use the same P expression in previous example to demonstrate the COM rule.*
Using OUT rule we can have following;
$\bar{s}a.\bar{s}b.\mathbf{0} \xrightarrow{\bar{s}a} \bar{s}b.\mathbf{0}$
Then by PAR rule we have following outcome; $\bar{s}a.\bar{s}b.\mathbf{0}|z(t).\mathbf{0} \xrightarrow{\bar{s}a} \bar{s}b.\mathbf{0}$
Further using INP we get following;
$s(v).s(u).\bar{v}u.\mathbf{0} \xrightarrow{s(a)} s(u).\bar{a}u.\mathbf{0}$
Applying COM (using above results)
$(\bar{s}a.\bar{s}b.\mathbf{0}|z(t).\mathbf{0})|s(v).s(u).\bar{v}u.\mathbf{0} \xrightarrow{\tau} (\bar{s}b.\mathbf{0}|z(t).\mathbf{0})|s(u).\bar{a}u.\mathbf{0}$
Then using RES we have following;
$P \xrightarrow{\tau} (\nu s)((\bar{s}b.\mathbf{0}|z(t).\mathbf{0})|s(u).\bar{a}u.\mathbf{0})$

Above we encoded operational semantics to π-calculus. Important aspect now is to check whether two processes are equivalent in their behaviour. To check whether 2 processes behaviorally equal we use a technique called *Bisimulation*. In next section we discuss about bisimulations in detail.

IV. Behavioural Equivalence

In most process algebras the equivalence of two processes is decided by building an equivalence relation on agents. π-calculus follows the same strategy. The basic technique behind behavioural equivalence is Bisimulation. In following we give the most general form of Bisimulation definition.

Definition 11. ***(Bisimulation)*** *Bisimulation is a symmetric binary relation $\mathcal{R}$ on agents satisfying*
$P\mathcal{R}Q$ *and* $P \xrightarrow{\alpha}$ *implies* $\exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$.

The basic intuition is that if P can do an action then Q can do the same action and the derivatives lie in the same relation.

In the context of π-calculus the most simplest form of Bisimulation is defined based on reduction relation. The reduction based bisimilarity is defined as follows;

Definition 12. ***(Reduction Bisimilarity)*** *A symmetric relation $\mathcal{S}$ is a reduction bisimulation if whenever $(P, Q) \in \mathcal{S}$ then $P \xrightarrow{\tau} P'$ implies $Q \xrightarrow{\tau} Q'$ for some Q' with $(P', Q') \in \mathcal{S}$. Reduction Bisimilarity is the union of all reduction bisimulations.*

Reduction Bisimilarity is not the best process equivalence for π-calculus. As an example consider following processes $\bar{x}y.\mathbf{0}$ and $\mathbf{0}$. These two processes does not depict the same execution behaviour, but reduction bisimilarity relates those two processes as they dont have τ transitions. Therefore reduction bisimilarity is a weak equivalence.

To obtain a satisfactory notion for equivalence we need to observe processes in more detail. There are several plausible bisimilarities on π-calculus based on direct comparison of the actions that they can perform. Each of them has different qualities. In rest of the section we discuss about the most natural form of bisimilarity known as *Strong Bisimilarity*.

A. Strong Bisimilarity

The Strong Bisimilarity definition is based on general bisimulation definition, but with special care to define simulation rule for input actions. In following we discuss why we need special care in defining strong bisimilarity with an example.

Consider processes $P = a(u)$ and $Q = a(x).(\nu v)\bar{v}u$. Intuitively both P and Q represent the same behaviour; i.e. they both can take an input via a and then terminate. But in Q the name u is free where u is bound in P. Therefore the x in Q cannot be α-converted to u. Further the transition $P \xrightarrow{a(u)} \mathbf{0}$ cannot be simulated by Q. But we can very well α-conver u in P and get the same behaviour between P and Q. Therefore the conclusion is it is sufficient for Q to simulate only the bound actions where the bound object is not free in Q. Therefore for the strong bisimulation definition we assume $bn(\alpha) \cap (fn(P) \cup fn(Q)$ is *empty*. This argument applies to both input and bound output actions.

In addition the input actions mean that the bound object is a placeholder for something to be received. Therefore, if $P \xrightarrow{a(x)} P'$ then the behaviour of P' must be considered for all substitutions $\{u/x\}$, and we must require that for each substitution Q' is related to P'.

We now define strong bisimulation by considering above discussed factors.

Definition 13. ***(Strong Bisimulation)*** *A strong bisimulation is a symmetric binary relation $\mathcal{R}$ on agents satisfying the following;*

$P\mathcal{R}Q$ and $P \xrightarrow{\alpha} P'$ where $bn(\alpha) \cap (fn(P) \cup fn(Q)) = \emptyset$ implies

1) *If $\alpha = a(x)$ then $\exists\ Q' : Q \xrightarrow{a(x)} Q' \wedge \forall u : P'\{u/x\}\mathcal{R}Q'\{u/x\}$*
2) *If α is not an input then $\exists\ Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$*

P and Q are strongly bisimilar, written $P \sim Q$, if they are related by a bisimulation.

It follows that $\sim$ is the union of all bisimulations. Further we also have $P \equiv Q$ implying $P \sim Q$ (Using STRUCT rule in Definition 9). In following we will go through few examples.

Bisimulation is like a two player game on a directed graph where nodes represent processes and each directed edge represent a transition given in Table 4. The players move alternately. A play begins with two nodes occupied by tokens. The first player can move either of the tokens from the node along an outgoing edge to a neighboring node. The second player responds only by moving the remaining token from the node it is on along an outgoing edge which had the same action as first player's. If play is infinite then the second player wins, if after some finite number of moves the player whose turn it is, cannot move, then that player loses. To check whether 2 processes are in a bisimulation we apply this gaming strategy. First we will assume two processes are in a bisimulation, then we execute transition from Table 4, at a time we apply the same transition to both processes. After some finite number of steps if one process has a transition and if other doesn't for same action then we decide two processes are not related by a bisimulation.

Example 22. *Consider $P_1 = a(x).P + a(x).\mathbf{0}$ and $P_2 = a(x).P + a(x).if\ x = u\ then\ P$*
Assume $P \nsim \mathbf{0}$. Then is $P_1 \sim P_2$?
Using INP rule we have following;

$$\frac{}{a(x).\mathbf{0} \xrightarrow{a(x)} \mathbf{0}}$$

Further using SUM rule we can derive following;

$$\frac{a(x).\mathbf{0} \xrightarrow{a(x)} \mathbf{0}}{P_1 \xrightarrow{a(x)} \mathbf{0}}$$

Similarly from P_2 we can either get $P_2 \xrightarrow{a(x)} P$ or $P_2 \xrightarrow{a(x)} if\ x = u\ then\ P$.
When $P_2 \xrightarrow{a(x)} P$ definitely $P_1 \nsim P_2$ as $P \nsim \mathbf{0}$.
Now consider when $P_2 \xrightarrow{a(x)} if\ x = u\ then\ P$. This also does not satisfy because when input is $a(u)$, $P_2 \xrightarrow{a(u)} P$ and $P \nsim \mathbf{0}$.
Therefore we can conclude $P_1 \nsim P_2$.

Example 23. *Consider $P_1 \overset{def}{=} (\nu z)(\bar{z}a|z(w).\bar{x}w)$ and $P_2 \overset{def}{=} \tau.\bar{x}a$. We need to find whether $P_1 \sim P_2$.*
Suppose $P_1 \sim P_2$. Now lets look at possible actions we can perform on P_1. The only possible action is internal action. Therefore we have $P_1 \xrightarrow{\tau} (\nu z)(\mathbf{0}|\bar{x}a)$
Let's check how P_2 reacts to τ; $P_2 \xrightarrow{\tau} \bar{x}a$.
$\therefore (\nu z)(\mathbf{0}|\bar{x}a) \sim \bar{x}a$
Similarly we can prove $(\nu z)(\mathbf{0}|\bar{x}a) \xrightarrow{\bar{x}a} \mathbf{0}$
Further we have $\bar{x}a(\equiv \bar{x}a.\mathbf{0}) \xrightarrow{\bar{x}a} \mathbf{0}$.
We know $\mathbf{0} \sim \mathbf{0}$ (because they are structurally same), therefore our initial assumption is correct and $P_1 \sim P_2$.

We are tempted to think if two processes are bisimilar then the processes we get by prefixing an input are also bisimilar. However previous statement is not true. We prove this using a counter example.

Example 24. *Let $P_1 \overset{def}{=} \bar{z}x.\mathbf{0}|a(x).\mathbf{0}$ and $P_2 \overset{def}{=} \bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0}$*
Suppose $P_1 \sim P_2$. P_1 has 2 possible actions. They are to receive an input via a or send a name via z.
We will first consider sending a name via z. Then we have following (using PAR rule);

$$\frac{\bar{z}x.\mathbf{0} \xrightarrow{\bar{z}x} \mathbf{0}}{\bar{z}x.\mathbf{0}|a(x) \xrightarrow{\bar{z}x} \mathbf{0}}$$

Now we perform the same action on P_2. Then we get following using SUM rule;

$$\frac{\bar{z}x.a(y).\mathbf{0} \xrightarrow{\bar{z}x} a(y).\mathbf{0}}{(\bar{z}.a(x).\mathbf{0} + a(x).\bar{z}y.\mathbf{0}) \xrightarrow{\bar{z}x} a(y).\mathbf{0}}.$$

Therefore $\mathbf{0} \sim a(y).\mathbf{0}$. Further we get $a(y).\mathbf{0} \xrightarrow{a(u)} \mathbf{0}$ and $\mathbf{0} \xrightarrow{a(u)} \mathbf{0}$ (input on an inactive process will remain inactive). Finally $\mathbf{0} \sim \mathbf{0}$.
We get similar result when action $a(u)$ is performed on P_1. Therefore $P_1 \sim P_2$.

Example 24 shows that $\bar{z}x.\mathbf{0}|a(x).\mathbf{0} \sim \bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0}$. Using that result can we say input prefixed processes of those are also strongly bisimilar ? i.e. $x(z).(\bar{z}x.\mathbf{0}|a(x).\mathbf{0}) \sim x(z).(\bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0})$? It turns out the answer is *no*. One name that can be received for both processes is $x(a)$. In that case after INP transitions we will have processes $\bar{a}x.\mathbf{0}|a(y).\mathbf{0}$ and $\bar{a}x.a(y).\mathbf{0} + a(z).\bar{a}t.\mathbf{0}$. Only process $\bar{a}x.\mathbf{0}|a(y).\mathbf{0}$ is capable of having a τ transition and other process cannot have a τ transition. Therefore input prefixed processes of strong bisimilar processes are not necessarily bisimilar.

Further generalized observation is strong bisimilarity is not preserved by substitution. i.e,

$$(\bar{z}x.\mathbf{0}|a(x).\mathbf{0})\{a/z\} \nsim (\bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0})\{a/z\}$$

To define a congruence relation that works for substitution we need to formulate a more stronger version of bisimilarity. In following we define *Strong full bisimilarity*.

Definition 14. ***(Strong full bisimilarity)*** *Processes P and Q are strong full bisimilar, $P \sim^c Q$ if $P\sigma \sim Q\sigma$ for every substitution σ.*

Strong full bisimilarity is strictly finer than strong bisimilarity. As per above example $\bar{z}x.\mathbf{0}|a(x).\mathbf{0}$ and $\bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0}$ are strong bisimilar but *not* strong full bisimilar as substitution $\{a/z\}$ does not generate strong bisimilar processes.

A variation of Example 24 that is full bisimilar is as follows;
Let $M \overset{\text{def}}{=} \bar{z}x.\mathbf{0}|a(x).\mathbf{0}$ and $N \overset{\text{def}}{=} (\bar{z}x.a(y).\mathbf{0} + a(x).\bar{z}y.\mathbf{0} + (if\ z = a\ then\ \tau.\mathbf{0})$
Then; $M \sim^c N$. The reason is N now has a if condition. This if condition helps to keep terms bisimilar when they are input prefixed and when input name is same as "a".

We now establish an important theorem in behavioural equivalence for π-calculus.

Theorem 1. ***(Congruence for*** $\sim^c$***)*** $\sim^c$ *is a congruence.*

[6] gives the detail proof for above theorem. In following we will go through sketch of the proof.

Consider processes P and Q. Suppose $P \sim^c Q$. We do induction on contexts, i.e. for every context C and substitution σ, $C[P\sigma] \sim^c C[Q\sigma]$. In the base case of induction, i.e. when $C = [.]$ the result follows immediately as $P \sim^c Q$. Then we perform induction for other cases of contexts based on initial results.

We now have established a mechanism to check whether 2 processes behave in the same way for all inputs. Technically to check whether 2 processes are behaving same we need to check whether the process pair is in a strong bisimulation. Theorem 1 guarantees that both processes have the same behaviour on all substitutions.

V. Variations of π-Calculus

The core of the π-calculus consists of a syntax definition, semantic definition, actions and notion of a behavioural equivalence. There are several variations of π-calculus where those variations alter sections associated with core π-calculus. These variations are used in different domains.

The calculus we learnt so far is sometimes referred to *monadic* π-calculus, because an interaction involves the communication of a single name between processes. *Polyadic* calculus can communicate multiple names in a form of tuples. In polyadic calculus outputs look like $\bar{a} < y_1, ...y_n >$ and inputs are of type $a(x_1, ...x_n)$. An interesting question in polyadic calculus is how to reduce an expression like $a(xy).P|\bar{a} < u > .Q$, where the arity of the output is not the same as the arity of the input. To solve this complication polyadic calculus assign some type of information to each name. In polyadic calculus calls that name a *sort*. The semantics for polyadic calculus is only notational more complex compared to monadic calculus.

Another important addition to $\pi - calculus$ is recursion. Recursion is a mechanism for describing iterative or arbitrarily long behaviours. In a way replication helps to define such behaviour. [6] shows that any process that involves recursion can be represented using replication, conversely whenever process involves replication we can use recursion to represent the program. Further there are several different variants of bisimulation defined for π-calculus: ground bisimilarity, open bisimilarity, late bisimilarity are some of such definitions. We will not go into details of those definitions. $Security\ \pi - calculus$ (spi) is an extension designed for the description and analysis of cryptographic protocols.

The $\pi - calculus$ we discussed so far is based on *synchronous* communication; i.e. one component emits a name and at the same time as another component receives it. An interesting variation of π-calculus is *asynchronous* π-calculus. Asynchronous π-calculus is particular important in modeling distributed systems. Therefore in next section we will discuss asynchronous π-calculus in more detail.

VI. Distributed Computing & π-Calculus

Distributed Systems involve communication through a network channel. Unlike in concurrent processes, processes in a distributed system have a delay associated with communication. Therefore the distributed communication can be treated as an asynchronous communication (Runtime environment or programming model may try to abstract communication as synchronous but whenever there is communication between a remote process the communication is asynchronous). Therefore the asynchronous π-calculus plays a major role in modeling distributed systems.

A. The Asynchronous π-Calculus ($A\pi$)

In asynchronous communication there is an unpredictable delay between output and input, especially when there is a message in a network channel (transit). This behaviour can be modeled by inserting an agent representing an asynchronous communication medium between sender and receiver. The properties of the medium (whether it has a bound on capacity, etc.) is then determined by the definition. An example is given below;

$$M \overset{\text{def}}{=} i(x).(\bar{o}x|M)$$

In above example when M receives a name u along i it evolves to $\bar{o}u|M$, and deliver u along o at any time and also continue to accept more messages.

$A\pi$ captures above described asynchronous communication in a form of a subcalculus. $A\pi$ consists agents satisfying the following requirements;

1) Only **0** can follow an output Prefix
2) An output Prefix may not occur as an *unguarded* operand of +

First of all we will focus on what we meant by *unguarded* above. In following we define *guarded* and *unguarded* process variables.

Definition 15. ***(Guarded vs Unguarded)*** *An occurrence of process variable X is guarded in term P if it occurs in Q where $\alpha.Q$ is a subterm of P.*
Process variable is unguarded when it is not guarded.

We now go back to agent requirements of $A\pi$. We will first consider second requirement. The second requirement basically disallows expressions like $\bar{a}x + b(y)$, but allows $\tau.\bar{a}x + b(y)$. The first requirement disallows agents such as $\bar{a}x.\bar{b}y$, where an agent other than **0** follows $\bar{a}x$.

The motivation to have above constructs is as follows; An unguarded output Prefix $\bar{a}x$ occurring in a term represents a message that has been sent but not yet received. The action of sending message is placed at an unguarded position as follows;

$$\tau.(\bar{a}x|P) \xrightarrow{\tau} \bar{a}x|P$$

After above transition, $\bar{a}x$ can interact with a receiver, and the sender proceeds concurrently as P. The requirement 1, forces sender to detect that message has been sent until receiver sends an acknowledgement. In summary the term $\tau.(\bar{a}x|P)$ can be read as "send $\bar{a}x$ asynchronously and continue as P".

B. Distributed Asynchronous PI-CALCULUS (ADPI)

Mathew Hennessy describes a variation of $A\pi$ specifically taking distributed computing concepts into consideration (cited in [8]). A distributed system formulated as a set of independent domains. A domain may have one or more processes running. ADPI augments $A\pi$ to add distributed system specific constructs. An interesting addition is the *process migration*. Process migration represents executing a process remotely. As per ADPI the process migration is depicted as follows;

$$gotok.P$$

Intuitively executing above statement in a domain l will migrate to location k and process P is launched.

VII. Programming Languages & π-Calculus

Type systems for Programming Languages are closely related with π-calculus. Type systems help to detect errors statically, improve the efficiency of code generated by a compiler, let users understand programs easily. *Type π-calculus* is a variation of $\pi-calculus$ that is closely related to programming language type system.

In Type π-calculus the programming language type system is formalized by means of *typing rules*. The terms that can be typed using these rules are called *well typed* terms. Fundamental property of the type system is its agreement with the reduction relation of the calculus. Accurately implemented type system guarantees that well typed terms do not raise runtime errors during reductions. This notion is also called *type soundness*.

In the literature we find several programming language implementations based on π-calculus. *Business Process Modeling Language*(BPML) [10] is a widely used workflow modeling language that is based on π-calculus. *occam-pi* [11] is a programming language that aims to write correct, expressive concurrent programs. *Pict* [9] is another experimental statically typed programming language that is based on π-calculus.

VIII. Summary

π-Calculus is a latest addition to the process algebra family. π-Calculus is different from other process algebra's due to its ability to send receive channels. In this paper we went through core π-calculus definitions with several examples. Definition of a π-calculus mainly consists of syntax definition, semantic (reduction) definition, congruence definition (both structural and behavioural). In this paper we went through each section with elaborating examples.

There are several variations of π-calculus. Most of the extensions are created by modifying the syntax, reduction behaviour, or by changing behavioural equivalence. Asynchronous π-calculus plays a major role in modeling distributed systems. Further π-calculus based programming languages guarantee significant safety during runtime. i.e. programming errors related to concurrency or distributed computing are captured at early stage (during compile) of the program.

In the literature already there are few experimental languages that are based on π-calculus (pl-pic). Those languages are introduced to the reader in the previous section.

π-calculus is highly relevant in building distributed systems. Further π-calculus is closely related to programming language type theory. In this paper we discussed about general π-calculus in depth with several examples. Finally we discussed how π-calculus is relevant to programming language research and distributed computing research.

References

[1] J. Bergstra and J. W. Klop, "Algebra of communicating processes," *Mathematics and Computer Science, CWI Monograph*, vol. 1, pp. 89–138, 1986.

[2] J. C. Baeten and W. P. Weijland, "Process algebra, volume 18 of cambridge tracts in theoretical computer science," 1990.

[3] C. A. R. Hoare, *Communicating sequential processes*. Prentice-hall Englewood Cliffs, 1985, vol. 178.

[4] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[5] J. A. Bergstra, A. Ponse, and S. A. Smolka, *Handbook of process algebra*. Elsevier, 2001.

[6] D. Sangiorgi and D. Walker, *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[7] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[8] M. Hennessy, *A distributed Pi-calculus*. Cambridge University Press, 2007.

[9] B. C. Pierce and D. N. Turner, "Pict: a programming language based on the pi-calculus." in *Proof, language, and interaction*, 2000, pp. 455–494.

[10] H. Smith, "Business process managementthe third wave: business process modelling language (bpml) and its pi-calculus foundations," *Information and Software Technology*, vol. 45, no. 15, pp. 1065–1069, 2003.

[11] P. H. Welch and F. R. Barnes, "Communicating mobile processes: introducing occam-pi. in 25 years of csp, volume 3525 of," *Lecture Notes in Computer Science*, pp. 175–210.