Imperial College
London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Concurrency

Author:
W (CID: your college-id number)

Date: October 24, 2019

1 Asynchronous π Calculus

The π-calculus is a mathematical model of processes whose interconnections change as they interact. The basic computational step is the transfer of a communication link between two processes; the recipient can then use the link for further interaction with other parties.

Suppose there is a server that communicates with a client and a printer. The server communicates with the printer using communication link a, and with the client using communication link b. The server passes access right of the printer, a, to the client such that the client can communicate with the printer.

In π-calculus, $\bar{b}\langle a\rangle$ means the server sends a along b to the client. The client received a channel name from b, then use the channel to pass data d to the printer, expressed as $b(x).\bar{x}d$.

In this chapter we focus on **asynchronous π-calculus**, which is a subset of π-calculus. In asynchronous communication, the output process $\bar{b}\langle a\rangle$ is a message in the communication layer waiting to be picked up by a receiver. There can be multiple communication layer at the same time and their order is not preserved.

1.1 Basic Definitions

Two major concepts are **channel names** and **variables**. Channel names are constants that represents one specific channel, usually denoted using $a, b, c \in \mathcal{N}$. Variables, denoted as $x, y, z \in \mathcal{V}$, are used as placeholders for received channel names.

To define **processes**, we use **identifiers**, u, v, to mean either channel name or variable. A process P, Q can be any of the following,

1. $\bar{u}\langle v\rangle$ a channel v is sent along channel u.
2. $u(x).P$ a channel x is received from channel u, then continue process P using the received channel.
3. $\mathbf{0}$ does not perform any action.
4. $P \mid Q$ represents the combined behaviour of P and Q executing in parallel.
5. $!P$ means infinite parallel composition, i.e. $P \mid P \mid P \mid \ldots$
6. $(\nu a)P$ is a restriction. It means the agent can use channel a to communicate between components within P, but not outside P.

1.2 Free Variables

Given a process P, the free variables of the process $f_v(P)$ is defined as follow:

$$f_v(x) = \{x\} \qquad\qquad f_v((\nu a)P) = f_v(P)$$

$$f_v(a) = \emptyset \qquad f_v(!P) = f_v(P)$$
$$f_v(\mathbf{0}) = \emptyset \qquad f_v(\bar{u}\langle v\rangle) = f_v(u) \cup f_v(v)$$
$$f_v(P \mid Q) = f_v(P) \cup f_v(Q) \qquad f_v(u(x).P) = f_v(u) \cup (f_v(P) \setminus \{x\})$$

1.3 Free Names

Given a process P, the free names of the process $f_n(P)$ is defined as follow:

$$f_n(x) = \emptyset \qquad f_n((\nu a)P) = f_n(P) \setminus \{a\}$$
$$f_n(a) = \{a\} \qquad f_n(!P) = f_n(P)$$
$$f_n(\mathbf{0}) = \emptyset \qquad f_n(\bar{u}\langle v\rangle) = f_n(u) \cup f_v(v)$$
$$f_n(P \mid Q) = f_n(P) \cup f_n(Q) \qquad f_n(u(x).P) = f_n(u) \cup f_n(P)$$

1.4 α-conversion

α-conversion is an operation to rename *bound* names and variables (not free). If P is obtained from Q using α-conversion, then they are α-equivalent, $P =_\alpha Q$.

If we want to rename a bound variable a to b, and b already exists in the original process. Then we must first rename b to something else before renaming a to b.

1.5 Substitution

Applying substitution to a process P, $P\{a \setminus x\}$, has the effect of replacing all **free** variable x to name a.

If the substitution clashes with one of the **bound names**, we must first use α-conversion to rename the clashed variable before substitution.

1.6 Structural Congruence

In π-calculus, there are many agents that are syntactically different but has the same behaviour. For instance $a(x).\bar{b}\langle x\rangle$ and $a(y).\bar{b}\langle y\rangle$, or $P \mid Q$ and $Q \mid P$. Structural Congruence is used to identify agents which intuitively represent the same thing.

If P and Q are α-equivalent then they are structural congruent

$$P =_\alpha Q \to P \equiv Q$$

The structural congruence operator is reflexive, symmetric, and transitive.

$$P \equiv P \qquad P \equiv Q \to Q \equiv P \qquad P \equiv Q \wedge Q \equiv R \to P \equiv R$$

If $P \equiv Q$, then the following are also true

$$(\nu a)P \equiv (\nu a)Q \qquad P \mid R \equiv Q \mid R$$
$$u(x).P \equiv u(x).Q \qquad !P \equiv !Q$$

1.6.1 Structural Congruence of Parallel Composition

The parallel composition operator is associative and commutative

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad\qquad P \mid Q \equiv Q \mid P$$

We can add and delete **0** processes at will

$$P \mid \mathbf{0} \equiv P$$

The Replication symbol can be fold and unfold as many times as we want

$$!P \equiv P \mid !P$$

1.6.2 Structural Congruence of Restriction

A restriction on a **0** processes is meaningless

$$(\nu a)\mathbf{0} \equiv \mathbf{0}$$

The order of adjacent restrictions does not matter

$$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$$

In fact, we can combine all adjacent restrictions into $(\nu a_1, \ldots, a_n)P$

We can open the scope given that it does not affect free variables in other processes

$$a \notin f_n(P) \rightarrow P \mid (\nu a)Q \equiv (\nu a)(P \mid Q)$$

1.7 Reduction

The Reduction relation describes how processes interact by exchanging messages. The simplest case of reduction is

$$\bar{a}\langle v \rangle \mid a(x).P \rightarrow P\{v/x\}$$

Which means a process sends v through channel a, another processes receives v from channel a then continue process P.

This can be extended to parallelism and restrictions. Given $P \rightarrow P'$, then

$$P \mid Q \rightarrow P' \mid Q \qquad \text{and} \qquad (\nu a)P \rightarrow (\nu a)P'$$

Given that $P \equiv Q$ and $P' \equiv Q'$ and $Q \rightarrow Q'$, then $P \rightarrow P'$

The reduction relation is **nondeterministic**. Depending on what we choose to reduce first may lead to a different result. For example

$$\bar{a}\langle b \rangle \mid \bar{a}\langle d \rangle \mid a(x).\bar{c}\langle x \rangle \rightarrow \bar{a}\langle b \rangle \mid \bar{c}\langle d \rangle$$
$$\bar{a}\langle b \rangle \mid \bar{a}\langle d \rangle \mid a(x).\bar{c}\langle x \rangle \rightarrow \bar{a}\langle d \rangle \mid \bar{c}\langle b \rangle$$

1.8 Atoms

Atoms are commonly used processes in name passing.

1.8.1 Fowarder

A Forwarder forwards messages receieved on channel a to channel b

$$FW\langle a,b\rangle \doteq a(v).\bar{b}\langle v\rangle$$

Multiple Fowarder can be chained together.

$$(\nu b)(FW\langle a,b\rangle \mid FW\langle b,c\rangle) \mid \bar{a}\langle d\rangle \rightarrow \bar{c}\langle d\rangle$$

(νb) is called the protector and it prevents interference of other processes. Without the restriction, the messages might not be forward successfully.

$$FW\langle a,b\rangle \mid FW\langle b,c\rangle \mid \bar{a}\langle d\rangle \mid b(x).\mathbf{0} \rightarrow \mathbf{0}$$

1.8.2 Duplicator

A Duplicator receives message on channel a and duplicate the message to b and c

$$D\langle a,b,c\rangle \doteq a(v).(\bar{b}\langle v\rangle \mid \bar{c}\langle v\rangle)$$

Multiple Duplicators can be chained together but must include a protector to prevent interference.

$$(\nu b)(D\langle a,b,c_1\rangle \mid D\langle b,c_2,c_3\rangle) \mid \bar{a}\langle d\rangle \rightarrow (\bar{c_1}\langle d\rangle \mid \bar{c_2}\langle d\rangle \mid \bar{c_3}\langle d\rangle)$$

1.8.3 Killer

A killer kills a message from channel a

$$D\langle a\rangle \doteq a(v).\mathbf{0}$$

1.8.4 Identity Receptor

An identity receptor receieves and forwards the message on channel a repeatedly.

$$I\langle a\rangle \doteq !FW\langle a,a\rangle$$

1.8.5 Equator

An equator for two channels a and b forwards a message between the channels repeatedly.

$$EQ\langle a,b\rangle \doteq !FW\langle a,b\rangle \mid !FW\langle b,a\rangle$$

1.8.6 Omega

A omega process continues infinite reductions by himself.

$$\Omega \doteq (\nu a)(!FW\langle a,a\rangle \mid \bar{a}\langle a\rangle)$$

The reduction process is therefore

$$\Omega \to \Omega \to \ldots$$

1.8.7 New Name Generator

A new name generator creates a new name when it is asked

$$NN\langle a\rangle \doteq !a(u).(\nu b)\bar{u}\langle b\rangle$$

For example

$$\begin{aligned} &\bar{a}\langle c\rangle \mid \bar{a}\langle d\rangle \mid NN\langle a\rangle \\ &\to (\nu b)\bar{c}\langle b\rangle \mid \bar{a}\langle d\rangle \mid NN\langle a\rangle \\ &\to (\nu b)\bar{c}\langle b\rangle \mid (\nu b')\bar{d}\langle b'\rangle \mid NN\langle a\rangle \end{aligned}$$

In the first reduction, NN generates a new name b and output to channel c. In the second reduction, since b is already used NN generates a new name b' and output to channel d.

2 Synchronous π Calculus

Recall in the previous section, we only looked at **Monadic Asynchronous π calculus**. Monadic means we are only sending one value through a channel and asynchronous means there is no continuation on the output process.

In this chapter we look at the full π-calculus. We first look at **Monadic Synchronous π calculus** then we later look at **Polyadic Synchronous π calculus**.

In the following sections, we will use $[\![P]\!] = Q$ to represent a mapping from P to Q.

2.1 Monadic Synchronous π Calculus

Synchronous π calculus allows a continuation process after the output process, $\bar{u}\langle v\rangle.P$. The reduction rule is defined as

$$\bar{a}\langle v\rangle.P \mid a(x).Q \to P \mid Q\{v/x\}$$

It is possible to encode Monadic Synchronous π calculus to Monadic Asynchronous π calculus.

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad\qquad [\![!P]\!] = ![\![P]\!]$$

$$
\begin{array}{ll}
[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] & [\![(\nu a)P]\!] = (\nu a)[\![P]\!] \\
[\![\bar{u}\langle v\rangle.P]\!] = (\nu c)(\bar{u}\langle c\rangle \mid c(y).(\bar{y}\langle v\rangle \mid [\![P]\!])) & y \notin f_v(P), c \notin f_n(P) \\
[\![u(x).P]\!] = u(y).(\nu d)(\bar{y}\langle d\rangle \mid d(x).[\![P]\!]) & y \notin f_v(P), d \notin f_n(P)
\end{array}
$$

The interesting case here is the communication between channels. In synchronous communication, two processes must first exchange channels before transmitting messages.

Notice the condition that y, c, d are not free variables or free names in P.

$$
\begin{aligned}
[\![\bar{b}\langle e\rangle.P]\!] \mid [\![b(x).Q]\!] &= (\nu c)(\bar{b}\langle c\rangle \mid c(y).(\bar{y}\langle e\rangle \mid [\![P]\!])) \mid b(y).(\nu d)(\bar{y}\langle d\rangle \mid d(x).[\![Q]\!]) \\
&\rightarrow (\nu c)(c(y).(\bar{y}\langle e\rangle \mid [\![P]\!])) \mid (\nu d)(\bar{c}\langle d\rangle \mid d(x).[\![Q]\!]) \\
&\rightarrow (\nu d)(\bar{d}\langle e\rangle \mid [\![P]\!] \mid d(x).[\![Q]\!]) \\
&\rightarrow [\![P]\!] \mid [\![Q]\!]\{e/x\}
\end{aligned}
$$

The communication between two processes is summarised as follow:

1. The **sender** creates a private channel c and send it along b.
2. The **receiver** receives c from b. Creates another private channel d and send it along c
3. The **sender** receives d from c and send message e along d
4. The **receiver** receives message e from d

2.2 Polyadic Synchronous π Calculus

Polyadic channels transmit multiple variables along the channel. In Polyadic Synchronous π Calculus, the reduction rule is

$$\bar{a}\langle v_1,\ v_2,\ \ldots,\ v_n\rangle.P \mid a(x_1,\ x_2,\ \ldots,\ x_n).Q \rightarrow P \mid Q\{\tilde{v}/\tilde{x}\}$$

We can encode Polyadic Synchronous π calculus to Monadic Synchronous π calculus for the communication processes.

$$
\begin{array}{ll}
[\![u(x_1,\ x_2,\ \ldots,\ x_n).P]\!] = u(z).z(x_1).z(x_2)\ldots z(x_n).[\![P]\!] & z \notin f_v(P) \\
[\![\bar{u}\langle v_1,\ v_2,\ \ldots,\ v_n\rangle.P]\!] = (\nu c)\bar{u}\langle c\rangle.\bar{c}\langle v_1\rangle.\bar{c}\langle v_2\rangle\ldots\bar{c}\langle v_n\rangle.[\![P]\!] & c \notin f_n(P)
\end{array}
$$

2.3 Branching and Selection

In Synchronous π-calculus, we extend the definition for processes to include two new behaviours. A process P can be any of the following:

1. $u \triangleright \{l_1 : P_1 \mathbin{[\!]} l_2 : P_2 \mathbin{[\!]} \cdots \mathbin{[\!]} l_n : P_n\}$ is branching. Each process $P_{1\ldots n}$ has a corresponding label $l_{1\ldots n}$. Which process to run depends on which label u selected.
2. $u \triangleleft l.P$ selects label l and continue with process P.

The reduction rule for Branching and Selection is as follow

$$a \triangleright \{l_1 : P_1 \,[]\, \cdots \,[]\, l_n : P_n\} \mid a \triangleleft l_k.P \to P_k \mid P \qquad 1 \leq k \leq n$$

We can encode Branching and Selection to Polyadic Synchronous π calculus

$$\begin{array}{ll}
[\![u \triangleright \{l_1 : P_1 \,[]\, l_2 : P_2\}]\!] = u(x).(\nu c_1, c_2)(\bar{x}\langle c_1, c_2\rangle \mid c_1.[\![P_1]\!] \mid c_2.[\![P_2]\!]) & x \notin f_v(P)\ c_1, c_2 \notin f_n(P) \\
[\![u \triangleleft l_1.P]\!] = (\nu c)(\bar{u}\langle c\rangle \mid c(z_1, z_2).\bar{z_1}.[\![P_1]\!]) & z_1, z_2 \notin f_v(P)\ c \notin f_n(P) \\
[\![u \triangleleft l_2.P]\!] = (\nu c)(\bar{u}\langle c\rangle \mid c(z_1, z_2).\bar{z_2}.[\![P_2]\!]) & z_1, z_2 \notin f_v(P)\ c \notin f_n(P)
\end{array}$$

2.4 Variable Agent

A variable agent stores a value and allows two operations: read and write.

$$Var\langle a, x\rangle \doteq a(z).z \triangleright \{read : \bar{z}\langle x\rangle.Var\langle a, x\rangle \,[]\, write : z(y).Var\langle a, y\rangle\}$$

A Reader can read the value of a variable agent.

$$Reader\langle a, c\rangle \doteq (\nu s)\bar{a}\langle s\rangle.s \triangleleft read.s(y).\bar{c}\langle y\rangle$$

A Writer can change the value stored in a variable agent.

$$Writer\langle a, x\rangle \doteq (\nu s)\bar{a}\langle s\rangle.s \triangleleft write.\bar{s}\langle x\rangle.\mathbf{0}$$