Chapter 3

The Asynchronous π-calculus

In this chapter, a short introduction to the asynchronous π-calculus is given. The asynchronous π-calculus is a variant of the π-calculus based on the idea that the messages are elementary processes that can be sent without any sequencing constraint. This chapter only focuses on the syntax and structural congruence and reaction relation of the calculus. A detailed study on asynchronous π-calculus can be found in [4, 33].

3.1 Introduction

The π-calculus is a process algebra specially suited for the description and analysis of concurrent systems with dynamic or evolving topology. Systems are specified in the π-calculus as collection of processes or agents which interact by means of links or names. The calculus allows direct expression of mobility, which is achieved by passing link names as arguments or objects of messages. When an agent receives a name, it can use this name as a subject for future transmissions, which allows an effective reconfiguration of the system. In fact, the calculus does not distinguish between names and data. This homogeneous treatment of names is used to construct a very simple but powerful calculus.

The π-calculus is available in two basic styles: the monadic π-calculus, where exactly one name is communicated at each synchronization, and the polyadic π-calculus, where zero or more names are communicated. We will use both monadic and polyadic asynchronous π-calculus together as mentioned in Figure 3.1. below. The basic concept behind the π-calculus is naming or reference. Names are the primary entities and they may refer to links or channel or any other kind of basic entity. Processes, sometimes referred as agents, are the other kind of entities. We let the letters X₁,...,Xₙ, x, y, y₁,...,yₙ range over the names. We also let π, π₁, π₂ range over agents or process expressions.
Chapter 3. The Asynchronous $\pi$-calculus

System: $S := Q^+$

Process Definition: $Q := A(x_1, .., x_n) = \pi$ (where $i \neq j \Rightarrow x_i \neq x_j$) ; $n >= 0$

Process: $\pi := \text{nil}$ Nil

| $\alpha . \pi$ Prefix |
| $x' < y_1, .., y_n, \text{nil}$ Async. Output ; $n >= 0$ |
| $\pi_1 \parallel \pi_2$ Parallel |
| $\pi_1 + \pi_2$ Sum |
| (new x) \pi Restriction |
| $[x=y] \pi$ Match |
| $[x!=y] \pi$ Mismatch |
| $A(x_1, .., x_n)$ Identifier/Process Instantiation $n >= 0$ |

Action Prefixes: $\alpha := x(y_1, .., y_n)$ Input

| $\tau$ Silent |

Figure 3.1 The syntax of the asynchronous $\pi$-calculus.

3.2 The Asynchronous $\pi$-calculus Syntax

In Figure 3.1, it is seen that a system in the $\pi$-calculus is a composition of one or more process definition(s), where process definition is of the form $A(X_1, .., X_n) = \pi$ and processes(agents) could have the following forms:

(1) $\text{nil}$ is an inactive or deadlock process; it is a process that can do nothing.

(2) The prefix $\alpha . \pi$ has a single capability, expressed by $\alpha$; the process $\pi$ cannot proceed until that capability has been exercised. The prefixes could be in two possible forms $x(y).\pi$ or $\tau . \pi$. “x(y)” is a positive prefix, where $x$ is the input port of an agent; it binds the variable $y$. At port $x$ the arbitrary name $z$ is input by $x(y).\pi$, which behaves like $\pi[y/z]$, where $\pi[y/z]$ is the result of substituting $z$ for all free (unbound) occurrences of $y$ in $\pi$. Similarly, arbitrary names $z_1, .., z_n$ are input by $x(y_1, .., y_n).\pi$ at port $x$ and behaves like $\pi[y_1/z_1, .., y_n/z_n]$. $\tau$ is the silent action; $\tau . \pi$ first performs the silent action and then acts like $\pi$.

(3) $x' < y, \text{nil}$ is an asynchronous output process. “x’<y>” is a negative prefix; x’ can be thought of an output port of an agent which contains it. $x' < y, \text{nil}$ outputs $y$ on port $x$ then behaves like $\text{nil}$ that can do nothing. We are using asynchronous $\pi$-calculus and therefore, only $\text{nil}$ could follow an output action. Similarly, $x' < y_1, .., y_n, \text{nil}$ outputs $y_1, .., y_n$ on port $x$ then behaves like $\text{nil}$.
Chapter 3. The Asynchronous $\pi$-calculus

(4) A parallel composition $\pi_1 || \pi_2$, which represents the combined behavior of $\pi_1$ and $\pi_2$ executing in parallel. The components of $\pi_1$ and $\pi_2$ can act independently, and may also communicate if one performs an output and the other an input along the same port.

(5) $\pi_1 + \pi_2$ represents sum-nondeterminism, that is, do either process $\pi_1$ or process $\pi_2$.

(6) (new x) $\pi$ means that x is declared as a new name local to process $\pi$ and is bound in $\pi$. It is not visible outside $\pi$.

(7) $[x=y] \pi$ represents the process that changes to $\pi$ if x=y. Mismatch is the opposite, i.e., it checks x!=y.

(8) $A(x_1,..,x_n)$ represents the instantiation of a defined agent.

(9) $A(x_1,..,x_n) = \pi$ (where i ≠ j => x_i ≠ x_j ) represents the declaration of a process $A$ in terms of process $\pi$. One can think of it as a procedure declaration in traditional procedural programming.

3.3 Free and Bound Occurrences of Names

The input prefix and the new operator bind the names. For example, in a process x(y). $\pi$, the name y is bound. In (new x)$\pi$, x is considered to be bound. Every other occurrences of a name like x in x(y).$\pi$ and x, y in x’<y>.$\pi$ are free.

3.4 Structural Congruence and Reaction

To simplify the definition of reaction relation, we first introduce a structural congruence between process expressions.

Let $P^{\pi}$ be set of process expressions and let $P, Q \in P^{\pi}$. Here P, Q are called structurally congruent, written as $P \equiv Q$, if one can be transferred into the other by,

(i) Renaming of bound names i.e. $\alpha$ -conversion.

(ii) Reordering of terms in a summation i.e., commutativity of “+” i.e.,

$P + Q \equiv Q + P, \ P + (Q + R) \equiv (P + Q) + R$

(iii) $P || Q = Q || P, \ P || (Q || R) = (P || Q) || R, \ P || \text{n}i\text{l} \equiv P$

(iv) (new x) (P || Q) $\equiv P ||(\text{new x}) Q$ if x $\notin \text{freeName}(P), \ (\text{new x}) \text{n}i\text{l} \equiv \text{n}i\text{l}$
The structural congruence is a congruence relation on $P^\pi$ i.e. if $P \equiv Q$ then,

(i) $\pi.P + R \equiv \pi.Q + R$

(ii) $P \parallel R \equiv Q \parallel R$ and $R \parallel P \equiv R \parallel Q$

(iii) (new x) $P \equiv$ (new x) $Q$

The reaction relation $=>$ $P^\pi$ is generated by the following rules:

$$\tau.P + Q \Rightarrow P^{(tau)}$$

$$\left(x' < z > \textit{nil} + S\right) \parallel (x(y).P + Q) \Rightarrow P[y/z] \parallel \textit{nil}^{(react)} \quad \text{(central rule)}$$

$$P \Rightarrow P' \quad \text{par}$$

$$P \parallel Q \Rightarrow P' \parallel Q' \quad \text{par}$$

$$\text{(new x) } P \Rightarrow \text{(new x) } P' \quad \text{res}$$

$$P \Rightarrow P' \quad \text{struct} \quad \text{if } P \equiv Q \quad \text{and } P' \equiv Q'$$

Here $P[y/z]$ denotes the replacement of every free occurrences of $y$ in $P$ by $z$.

**3.5 Scope Extrusion**

The power of the $\pi$-calculus arises from migrating local scopes. The *new* operator, introduced above, declares a local name, of which no other process is aware. When such a private link is passed to another process, it is called scope extrusion or migration. Such scope migration is the central feature of $\pi$-calculus and accounts for the modelling of mobility. Normally, we need to apply *react* rule and *congruence (iv)* for migrating the scope of the private link.

Let us illustrate *scope extrusion* with an example. We introduce a practical example to show how the $\pi$-calculus can be used.

Initially (Figure 3.2), only server process $S$ has access to printer process $P$ using a private link $a$. We suppose that $S$ also has a link $b$ to the client process $C$ and using link $b$, client $C$ can request server $S$ to get access to printer $P$. 
As per the request of the printer $P$, server $S$ is now willing to pass link $a$ (private to $S$ and $P$) to $C$. Let $S=b'<a>.S'$ and $C=b(y).C'$. Then the composite process is represented by

$$(\text{new } a) \ (b'<a>.S' \ || \ P) \ || \ b(y).C'$$

Here $\text{new } a$ represents the privacy of $S$’s link to $P$. After a communication along $b$ from $S$ to $C$ the process becomes

$$(\text{new } a) \ (S' \ || \ P \ || \ C' [y/a])$$

The resulting configuration is shown in Figure 3.3. As a result, the scope of channel $a$ has effectively migrated from $S$ to $C$.

**Figure 3.2** Server $S$ has access to printer $P$

**Figure 3.3** Client $C$ has access to printer $P$