Fakultät II: Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung: Entwicklung korrekter Systeme

# Transformational semantics of the combination $\pi$-OZ for mobile processes with data

Masterarbeit

*- post version -*

Name: Muhammad Ekbal Ahmad
E-Mail: muhammad.ekbal.ahmad@uni-oldenburg.de

Studiengang: Fach-Master Informatik
Erstgutachter: Prof. Dr. Ernst-Rüdiger Olderog
Zweitgutachter: M.Sc. Manuel Gieseking
Datum: 16. Juni 2020

# Contents

# List of Figures

# 1 Introduction

every entity has a behavior and data. behavior actions can have effects on data. to model this idea we will break down our entity into two components: behavior component and data component. behavior component: represent the behavior that can an entity do during it's life cycle. data component: represents the data of an entity and the changes that can be made on it. we use pi calculus which is a specification language to model the behavior component. we use oz which is a specification language to model the data components. since pi and oz are two different languages used to descripe different aspects of entity, we need a way to put them togher to get the model a complete entity. this is done using a simple trick. the trick is: transforming the oz into a pi language. this way we will have : behavior component: in pi. data component: in pi too. this way we can let them play together to represent an entity which have two view: behavior and data.

# 2 Preliminaries

## 2.1 The $\pi$-calculus

The $\pi$-*calculus* is a process algebra that can be used to describe a behavior. This section introduces the pure polyadic version of the $\pi$-calculus as decriped in [Mil99].

### 2.1.1 Intuition

To explain the $\pi$-calculus inuitivly we will use the ion example as in [Mil99]. Let us imagine a positive and a negative ion. When those two ions merge, we get a new construct. The merge operation is called a *reaction*, since an ion acts and the other reacts. This reaction can be seen as communication between two processes. The two processes communicate to share some information. One process is the sender and the other is the receiver. By doing the reaction both processes evoulte to some thing new. The reaction, information sharing and evolution concepts are the core of the $\pi$-calculus. Using those concepts we can understand the title of Milner's book *communicating and mobile processes: the* $\pi$-calculus [Mil99]. The word *communicating* refers to the *reaction* concept. The word *mobile* refers to the *information sharing and evolution* concepts, since the receiver process can use the received information to change its location as we will see in Section 2.1.5.
Intuitively, the $\pi$-calculus consists of:

- a set of names starting with capital case letters like $P, P_1, Q, ..etc$ used to refer to a process directly.
- a set of names starting with capital case letters like $A, B, C, ..etc$ used a process identifier. The process identifier will be used to define recursion with parameters.
- a set of names starting with lower case letter like $a, b, x, y, ..etc$ used as a channel and message name. This set is denoted by $\mathcal{N}$.

- operators like:
  - Parallel composition operator: “ | ”.
  - Sequential composition operator: “ . ”.
  - Choice operator: “ + ”.
  - Scope restriction operator: “ $\underline{\texttt{new}}$ ”.

So a simple example of a process can be: $\overline{x}\langle y\rangle.\mathbf{0}$ this process simply sends the message $y$ via the channel $x$ and stops. The full syntax of $\pi$-calculus process is given in Definition 2.1.1. In this thesis starting from this point, when we mention the word *names* we refer to $\mathcal{N}$. Furthermore, we shall often write $\vec{y}$ for a sequence $y_1, ...., y_n$ of names.

### 2.1.2 Syntax

**Definition 2.1.1 (Process syntax)** The syntax of a $\pi$-calculus process $P$ is defined by:

$$P ::= \sum_{i \in I} \pi_i.P_i \;\Big|\; P_1 \mid P_2 \;\Big|\; \underline{\texttt{new}}\,\vec{y}\;P \;\Big|\; A\langle\vec{v}\rangle$$

where:

- $\sum_{i \in I} \pi_i.P_i$ is the guarded sum.
- $P_1 \mid P_2$ is the parallel composition of processes.
- $\underline{\texttt{new}}\,\vec{y}\;P$ is the restriction of the scope of the names $\vec{y}$ to the process $P$
- $A\langle\vec{v}\rangle$ is a process call. △

***Guarded sum*:**

The guarded sum is the *choice* between multiple guarded processes. If the guard of one process took place, other guarded processes will be discarded. For example, the processes: $x().P_1 + y().P_2$ will evolve to the process $P_1$ if the guard $x()$ occurred.

Furthermore, The process $\mathbf{0}$ is called the *stop process* or *inaction* and stands for the process that can do nothing. It can be omitted.

### *Guard*:

The guard is also called *action prefix* and denoted by $\pi$. It's syntax is defined by:

**Definition 2.1.2 (Action prefix syntax)**

$$\pi ::= \overline{x}\langle\vec{y}\rangle \;\big|\; x(\vec{y}) \;\big|\; \tau$$

where:

- $\overline{x}\langle\vec{y}\rangle$[1] represents the action: send $\vec{y}$ via the channel $x$.
- $x(\vec{y})$[2] represents the action: receive $\vec{y}$ via the channel $x$.
- $\tau$ represents an internal non observable action. $\triangle$

The set of all *actions* is defined as $\texttt{Act} =_{\text{def}} \texttt{Out} \cup \texttt{In} \cup \{\tau\}$, where:

- $\texttt{Out}$ is the set of all *output actions*, defined as $\texttt{Out} =_{\text{def}} \{\overline{x}\langle\vec{y}\rangle \mid x \in \mathcal{N}\}$.
- $\texttt{In}$ the set of all *input actions*, defined as $\texttt{In} =_{\text{def}} \{x(\vec{y}) \mid x \in \mathcal{N}\}$.

### *Parallel composition*:

The parallel composition operator $|$ represents the concept of concurrency in the $\pi$-calculus, where two processes can evolve in concurrent.It represents an interleaving behavior of the concurrency. For example let: $P =_{\text{def}} P_1 \mid (P_2 \mid P_3)$ where: $P_1 =_{\text{def}} x(y).Q_1$ , $P_2 =_{\text{def}} \overline{x}\langle y\rangle.Q_2$ and $P_3 =_{\text{def}} x(y).Q_3$. So $P =_{\text{def}} x(y).Q_1 \mid (\overline{x}\langle y\rangle.Q_2 \mid x(y).Q_3)$. Possible evolution cases of $P$ are:

- $P_1 \mid (Q_2 \mid Q_3)$. $P_2$ sends $y$ via $x$ to $P_3$.
- $Q_1 \mid (Q_2 \mid P_3)$. $P_2$ sends $y$ via $x$ to $P_1$.

The example above illustrated the privacy nature of the parallel operator in the $\pi$-calculus. A process can via a channel communicate with only one process pro time, i.e., the channel represents a binary synchronization. $P_2$ cannot communicate with both $P_1$, $P_3$ in the same time, while in Communicating Sequential Processes (CSP) a process can communicate with multiple processes in the same time via the same channel by sending multiple copies of the same message, i.e., in CSP the channel represents a multiple synchronization.

[1] $\overline{x}\langle\rangle$ means: send a signal via $x$. $\overline{x}\langle y\rangle$ means: send the name $y$ via $x$. $\overline{x}\langle\vec{y}\rangle$ means: send the sequence $\vec{y}$ via $x$.

[2] $x()$ means: receive a signal via $x$. $x(y)$ means: receive any name $y$ via $x$. $x(\vec{y})$ means: receive any sequence $\vec{y}$ via $x$. "$y$ here plays the role of parameter"

***Restriction*:**

The expression $\underline{\texttt{new}}\,\vec{y}\ P$ binds the names $\vec{y}$ to the process $P$. In other words: the visibility scope of the names $\vec{y}$ is restricted to the process $P$. It is similar to declaring a private variable in programming languages. Thus the names $\vec{y}$ are not visible outside $P$ and $P$ cannot use them to communicate with outside. For example, let $P =_{\text{def}} P_1 \mid P_2$ where: $P_1 =_{\text{def}} \underline{\texttt{new}}\,y\ \overline{y}\langle z\rangle.Q_1$ and $P_2 =_{\text{def}} y(z).Q_2$. The process $P$ cannot evolute to $Q_1 \mid Q_2$, since the name $y$ in $P_1$ is only visible inside it, i.e., from the $P_2$'s point of view $P_1$ doesn't have a channel called $y$. This takes us to the definition of the Bound and free names.

**Definition 2.1.3 (Bound names)** are all the restricted names in a process. △

**Definition 2.1.4 (Free names)** are all the name that occur in a process except the bound names. △

For example, let $P_1 =_{\text{def}} \underline{\texttt{new}}\,x\ \overline{x}\langle y\rangle.P_2$ where $P_2 =_{\text{def}} \underline{\texttt{new}}\,z\ \overline{x}\langle z\rangle.P_3$. The name $x$ is bound in $P_1$ but free in $P_2$.

***Process call*:**

Let $P$ be a process and let $A$ be a process identifier.To be able to use the process $P$ recursively we use the process identifier $A$ as follow: $A(\vec{w}) =_{\text{def}} P$. Thus , when we write $A\langle\vec{v}\rangle$ we are using the identifier $A$ to call the process $P$ with replacing the names $\vec{w}$ in $P$ with the names $\vec{v}$. This replacement is called the $\alpha$-conversion

For example, let $P =_{\text{def}} \overline{w}\langle y\rangle.\mathbf{0}$ and let $A(w) =_{\text{def}} P$ be the recursive definition of the process $P$, then the behavior of $A\langle v\rangle$is equlivant to $\overline{v}\langle y\rangle.\mathbf{0}$

### 2.1.3 Semantics

To understand the operational semantics of $\pi$-calculus we will use a labelled transition system LTS. Using this LTS we can investigate $\pi$-calculus process evolution. The definition of LTS is adapted from [Mil99] pages 39[3], 91[4], 132[5] with some changes.

**Definition 2.1.5 (LTS of $\pi$-calculus)** The labelled transition system $(\mathcal{P}^\pi, \mathcal{T})$ of $\pi$-calculus processes over the action set `Act` has the process expressions $\mathcal{P}^\pi$ as its

[3]Transition Rules: LTS for concurrent processes not for $\pi$-calculus processes.
[4]Reaction Rules: no labels and no LTS.
[5]Commitment Rules: abstractions and concretions are out of this thesis's scope.

states, and its transitions $\mathcal{T}$ are those which can be inferred from the rules in Figure 2.1. The rule REACT is the most important one. It shows the process evolution when a reaction occurs. The reaction requires two complementary transitions $P \xrightarrow{\overline{x}\langle\vec{y}\rangle} P'$ and $Q \xrightarrow{x(\vec{z})} Q'$, we call them commitments. so the process $P$ takes a commitment to take part in the reaction, and so does $Q$.

$$\underline{\mathit{OUT}} : \overline{x}\langle\vec{y}\rangle.P \xrightarrow{\overline{x}\langle\vec{y}\rangle} P \qquad \underline{\mathit{IN}} : x(\vec{y}).P \xrightarrow{x(\vec{y})} P$$

$$\underline{\mathit{TAU}} : \tau.P \xrightarrow{\tau} P \qquad \underline{\mathit{SUM}} : \alpha.P + \sum_{i \in I} \pi_i.P_i \xrightarrow{\alpha} P$$

$$\underline{\mathit{L\_PAR}} : \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \underline{\mathit{R\_PAR}} : \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\underline{\mathit{RESTRICTION}} : \frac{P \xrightarrow{\alpha} P'}{\underline{\mathtt{new}}\, x\ P \xrightarrow{\alpha} \underline{\mathtt{new}}\, x\ P'} \quad \mathit{if}\ \ \alpha \notin \{\overline{x}, x\}$$

$$\underline{\mathit{PROCESS\_CALL}} : \frac{\{\vec{y}/\vec{z}\}\, P \xrightarrow{\alpha} P'}{A\langle\vec{y}\rangle \xrightarrow{\alpha} P'} \quad \mathit{if}\ \ A(\vec{z}) =_{\text{def}} P$$

$$\underline{\mathit{REACT}} : \frac{P \xrightarrow{\overline{x}\langle\vec{y}\rangle} P' \quad Q \xrightarrow{x(\vec{z})} Q'}{P \mid Q \xrightarrow{\tau} P' \mid \{\vec{y}/\vec{z}\}\, Q'} \qquad \triangle$$

Figure 2.1: The transition rules [Mil99].

An example of using the transition rules of this LTS to infer a transition is: Let $P =_{\text{def}} \underline{\mathtt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle)$, where: $A_1(y) =_{\text{def}} \overline{y}\langle\rangle.A_2\langle y\rangle$ and $B_1(z) =_{\text{def}} z().B_2\langle z\rangle$. $P$ can do the transition $\underline{\mathtt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle) \xrightarrow{\tau} \underline{\mathtt{new}}\, x\ (A_2\langle x\rangle \mid B_2\langle x\rangle)$, which is a reaction. The inference tree of this transition is shown in Figure 2.2. Thus, using the LTS we can enumerate sll possible transitions of a π-calculus process.

$$
\cfrac{
  \cfrac{
    \cfrac{}{\overline{x}\langle\rangle.A_2\langle x\rangle \xrightarrow{\overline{x}\langle\rangle} A_2\langle x\rangle}\ \text{by OUT}
  }{A_1\langle x\rangle \xrightarrow{\overline{x}\langle\rangle} A_2\langle x\rangle}\ \text{by PROCESS CALL}
  \qquad
  \cfrac{
    \cfrac{}{x().B_2\langle x\rangle \xrightarrow{x()} B_2\langle x\rangle}\ \text{by IN}
  }{B_1\langle x\rangle \xrightarrow{x()} B_2\langle x\rangle}\ \text{by PROCESS CALL}
}{
  \cfrac{A_1\langle x\rangle \mid B_1\langle x\rangle \xrightarrow{\tau} A_2\langle x\rangle \mid B_2\langle x\rangle}{\underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle) \xrightarrow{\tau} \underline{\texttt{new}}\, x\ (A_2\langle x\rangle \mid B_2\langle x\rangle)}\ \text{by RESTRICTION}
}\ \text{by REACT}
$$

Figure 2.2: The inference tree [Mil99].

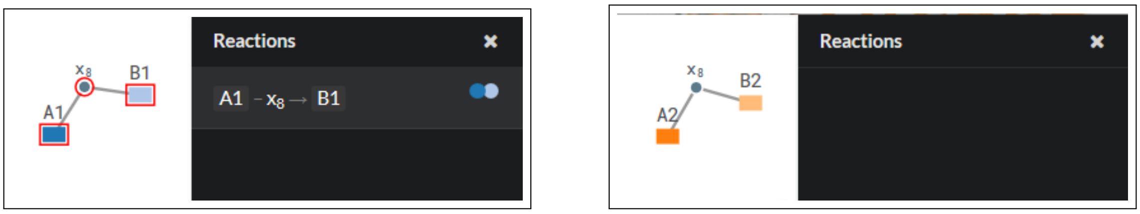
### 2.1.4 Visualization

To gain more understanding of the $\pi$-calculus we will use *Stargazer*[Star]. Stargazer is a visual simulator for $\pi$-calculus. Listing 2.1 shows the code of the process $P =_{\text{def}} \underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle)$ where: $A_1(y) =_{\text{def}} \overline{y}\langle\rangle.A_2\langle y\rangle$ and $B_1(z) =_{\text{def}} z().B_2\langle z\rangle$ in stargazer syntax.

```
new x. (A1[x] | B1[x])
A1[y] := y<>.A2[y]
B1[z] := z().B2[z]
```

Listing 2.1: stargazer code for the process $P$.

Stargazer can visualize the reaction $\underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle) \xrightarrow{\tau} \underline{\texttt{new}}\, x\ (A_2\langle x\rangle \mid B_2\langle x\rangle)$ as shown in Figure 2.3.




(a) Before reaction occurrence. (b) After reaction occurrence.

Figure 2.3: The process $P$ reaction

### 2.1.5 Mobility

As mentioned previously, the word *mobile* refers to the *information sharing and evolution* concepts, since the receiver process can use the received information to change its location. Let us take an example to illustrate the mobility. Let: $\underline{\texttt{new}}\, x, y\ (A\langle x, y\rangle \mid B\langle x\rangle)$ where:

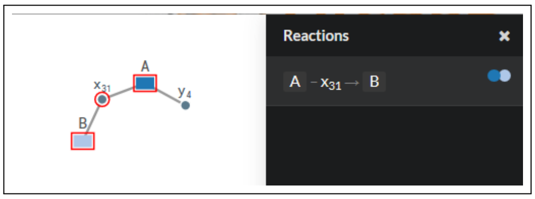
- $A(a, b) =_{\text{def}} \overline{a}\langle b\rangle.A\langle a, b\rangle$

- $B(c) =_{\text{def}} c(d).B\langle d\rangle$

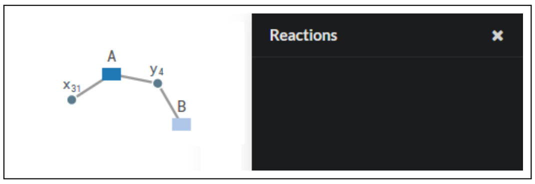
Listing 2.2 shows the stargazer code of the process $\underline{\texttt{new}}\, x, y\ (A\langle x, y\rangle \mid B\langle x\rangle)$, Figure 2.4 shows its visualization before and after the interaction occurrence.

```
new x,y. (A[x,y] | B[x])
A[a,b] := a<b>.A[a,b]
B[c] := c(d).B[d]
```

Listing 2.2: Stargazer code for the process $\underline{\texttt{new}}\, x, y\ (A\langle x, y\rangle \mid B\langle x\rangle)$.




(a) Before reaction occurrence.




(b) After reaction occurrence.

Figure 2.4: Mobility reaction

Intuitively, The mobility can be noticed in Figure 2.4, since $B$ changed it's position in the connection topology. The following explains the mobility through interaction step by step:

- Initially the process $A\langle x, y\rangle$ has the channels $x, y$ and the process $B\langle x\rangle$ has the channel $x$. Thus, $A\langle x, y\rangle$ and $B\langle x\rangle$ are connected via channel $x$.

- $A\langle x, y\rangle$ has commitment $\overline{x}\langle y\rangle$, i.e., send the channel name $y$ via the channel $x$.

- $B\langle x\rangle$ has commitment $x(d)$, i.e., receive a message $d$ via $x$.

- That means: a reaction can occur between $A\langle x, y\rangle$ and $B\langle x\rangle$. This reaction is: $\underline{\texttt{new}}\, x, y\ (\overline{x}\langle y\rangle.A\langle x, y\rangle \mid x(d).B\langle d\rangle) \xrightarrow{\tau} \underline{\texttt{new}}\, x, y\ (A\langle x, y\rangle \mid B\langle y\rangle)$.

- Information sharing: the process $A\langle x, y\rangle$ sends the name $y$ to $B\langle x\rangle$ when the interaction occurs.

- Evolution: when interaction occurs $B\langle x\rangle$ knows about the channel $y$ and uses it as parameter for the the process call $B\langle y\rangle$ .i.e, The $B\langle y\rangle$ now has the channel $y$, and no more $x$.

- Finally, in other words:
  - before the reaction: $B$ was connected to $A$ via $x$ as shown in Figure 2.4.
  - after the reaction: $B$ is connected to $A$ via $y$ as shown in Figure 2.4.

### 2.1.6 Strong simulation

The *strong simulation* is comparison of processes based on their behavior. To understand this let us start with a simple example: Let $P =_{\text{def}} \tau.\tau.\mathbf{0}$ and $Q =_{\text{def}} \tau.\mathbf{0}$. We can notice that $P$ can do two $\tau$ transitions, but $Q$ can do only one. Thus $Q$ doesn't strongly simulates $P$. The word *strongly* refers to the point that: the strong simulation comparison takes the internal transition $\tau$ into account. There is another kind of comparison called the *weak simulation*, which doesn't consider the internal transition $\tau$, but this kind of comparison is not considered in this thesis. The formal definition of the *strong simulation* is given in Definition 5.1.1, which is adapted from [Gi14] page 32 with some changes.

**Definition 2.1.6 (Strong simulation)** A relation $\mathcal{S} \subseteq \mathcal{P}^\pi \times \mathcal{P}^\pi$ is called a *strong simulation*, if $(P, Q) \in \mathcal{S}$ implies that

$$\text{if } P \xrightarrow{\alpha} P' \text{ then } Q' \in \mathcal{P}^\pi \text{ exists such that } Q \xrightarrow{\alpha} Q' \text{ and } (P', Q') \in \mathcal{S}. \qquad \triangle$$

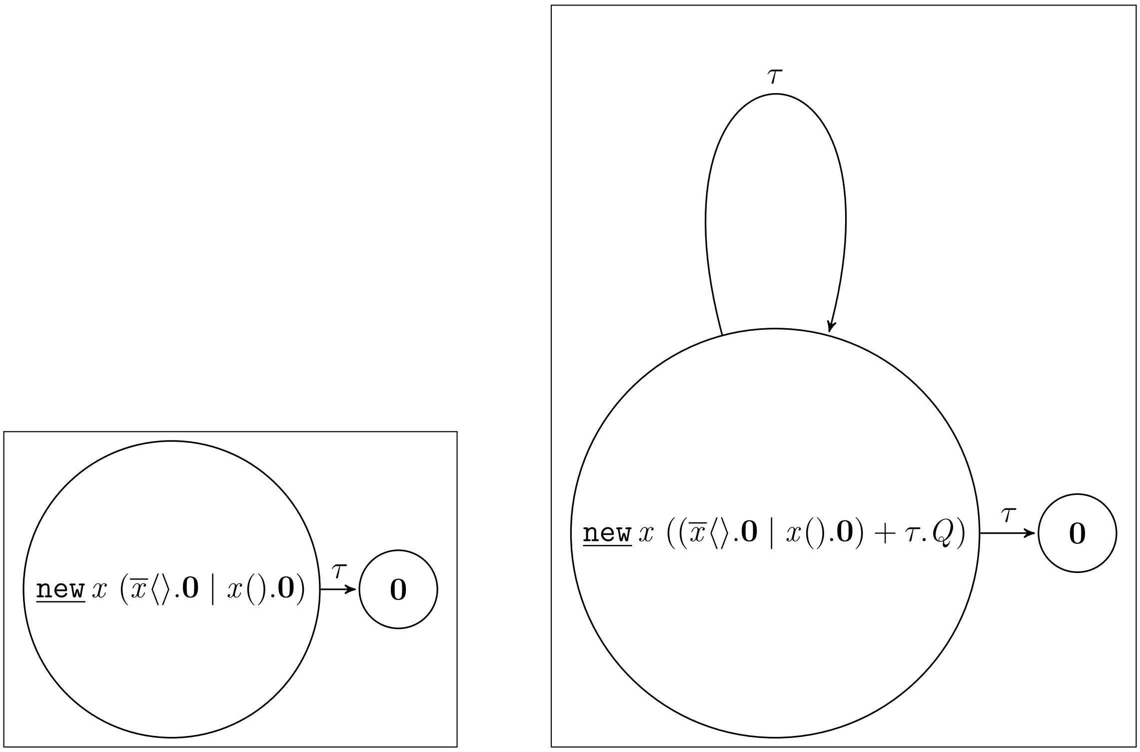
An example of checking the strong simulation is:
Let

- $P =_{\text{def}} \underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle)$
- $Q =_{\text{def}} \underline{\texttt{new}}\, x\ ((A_1\langle x\rangle \mid B_1\langle x\rangle) + \tau.Q)$

where:

- $A_1(y) =_{\text{def}} \overline{y}\langle\rangle.\mathbf{0}$
- $B_1(z) =_{\text{def}} z().\mathbf{0}$

Intuitively, The behavior of $P$ and $Q$ can be illustrated using transition graphs as shown in Figure 2.5. $Q$'s transition graph is the same as $P$'s, except one thing: $Q$ has a loop with label $\tau$. This loop is due to the $\tau$ transition in $Q$'s definition. Hence, we can notice that $Q$ can do all the transitions that $P$ can, plus an extra transition $\tau$. In other words $Q$ simulates $P$, but $P$ doesn't simulate $Q$.




(a) $P$

(b) $Q$

Figure 2.5: Transition graphs

To check the strong simulation we can use *ABC (Another Bisimilarity Checker)* [ABC]. ABC is a tool that checks simulation between $\pi$-calculus processes. Listing 2.3 shows the code of the process $P$ and $Q$ in ABC syntax.

```
agent P = (^x)( A_1 x | B_1 x)
agent A_1(y) = 'y.0
agent B_1(z) = z.0
agent Q = (^x)((A_1 x | B_1 x) + t.Q)
// check if Q strongly simulates P
lt P Q
// check if P strongly simulates Q
lt Q P
```

Listing 2.3: ABC code for $P$ and $Q$.

Listing 2.4 and Listing 2.5 shows the result of running Figure 2.3, where x0 stands for x, since ABC renames the channels and messages names internally.

In Listing 2.4 we see the result of the command *lt* $P$ $Q$, which checks if $Q$ strongly simulates $P$. The result is *yes* and the simulation relation is shown, where x0 stands for x. In Figure 2.4 we see the two pairs of the simulation relation, where:

- (0 { } 0) stands for the pair $(\mathbf{0}, \mathbf{0})$, which means: The state $\mathbf{0}$ of $Q$ is as powerful as $\mathbf{0}$ of $P$.
- ( (^x0)('x0.0 | x0.0) { } (^x0)(('x0.0 | x0.0) + t.Q) ) stands for the pair $(\underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle), \underline{\texttt{new}}\, x\ ((A_1\langle x\rangle \mid B_1\langle x\rangle) + \tau.Q))$, which means: The state $\underline{\texttt{new}}\, x\ ((\overline{x}\langle\rangle.\mathbf{0} \mid x().\mathbf{0}) + \tau.Q)$ of $Q$ is as powerful as $\underline{\texttt{new}}\, x\ (\overline{x}\langle\rangle.\mathbf{0} \mid x().\mathbf{0})$ of $P$.

Thus, $Q$ strongly simulates the behavior of $P$ and the simulation relation is $\mathcal{S} = \{(\mathbf{0}, \mathbf{0}), (\underline{\texttt{new}}\, x\ (A_1\langle x\rangle \mid B_1\langle x\rangle), \underline{\texttt{new}}\, x\ ((A_1\langle x\rangle \mid B_1\langle x\rangle) + \tau.Q))\}$.

```
The two agents are strongly related (2).
Do you want to see the core of the simulation (yes/no) ? yes
{
  (
    0
    { }
    0
  )

  (
    (^x0)('x0.0 | x0.0)
    { }
    (^x0)(('x0.0 | x0.0) + t.Q)
  )
}
```

Listing 2.4: ABC output: check if $Q$ strongly simulates $P$.

In Listing 2.5 we can see the result of the command *lt Q P*, which checks if $P$ strongly simulates $Q$. The result is *no*, since:

- when:
  - $Q$ is in the state $\underline{\texttt{new}}\, x\ ((\overline{x}\langle\rangle.\mathbf{0} \mid x().\mathbf{0}) + \tau.Q)$.
  - $P$ is in the state $\underline{\texttt{new}}\, x\ (\overline{x}\langle\rangle.\mathbf{0} \mid x().\mathbf{0})$.
- then:

  - $Q$ can do a $\tau$ transition, which is the loop, to the state $\underline{\texttt{new}}\, x\ ((\overline{x}\langle\rangle.\mathbf{0} \mid x().\mathbf{0}) + \tau.Q)$.
  - $P$ can do a $\tau$ transition, which is a reaction, to the state $\mathbf{0}$.

- then:
  - $Q$ can do a $\tau$ transition, which is a reaction, to the state $\mathbf{0}$.
  - $P$ cannot go ahead, denoted by " $*$ ", since it is in the state $\mathbf{0}$.

Thus, $P$ doesn't strongly simulates the behavior of $Q$.

```
The two agents are not strongly related (2).
Do you want to see some traces (yes/no) ? yes
traces of

Q
P

-t->
-t->

(^x0)(('x0.0 | x0.0) + t.Q)
0

-t->
-t->

0
*
```

Listing 2.5: ABC output: check if $P$ strongly simulates $Q$.

## 2.2 The Object-Z

The Object-Z, shortly OZ, is a specifications language used to describe an entity through specifing its data, operations and the effects of those operations on the data. This section introduces the Object-Z as decriped in [Ol18].

### 2.2.1 Intuition

To explain the OZ intuitively, we will start by examining the vending machine example, then we will explain how to build a set mathematically, finally we will explain the main concepts in OZ.

***Vending Machine*:**

As a preperation, let us imagine that we have the task: specifying a vending machine.

- Let $cv$ be the ammount of coffee, and $tv$ be the amount of tea.
- Let *coffee* be the selling coffee operation, and *tea* be the selling tea operation.

the specifications are:

- It sell *coffee* and *tea*, and the maximum amount for each if them is 3.
- It's initial state is $cv = 3$ and $tv = 3$.
- When the operation *coffee* or *tea*,then the amount should be decreased by one.

The state space of the vending machine can be visualized as shown in Figure 2.6, where we see the initial state $VM(3,3)$. The arrow indicates a state transition decrementing the amount of coffee. Later in **Main concepts of OZ** we will learn how to write the specifications using OZ language notations.

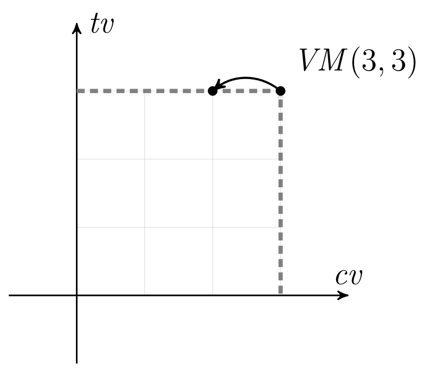



Figure 2.6: State Space.

***Set building*:**

A set is a collection of things. For example: $\{5, 7, 11\}$ is a set. But we can also build a set by describing what is in it using the following notation: $\{Deklaration \mid predicate \bullet expression\}$. For example: $\{x : \mathbb{Z} \mid x \geq 0 \bullet x^2\}$ means *the set of all squared $x$'s, such that $x$ is integer and greater than 0*

***Main concepts of OZ*:**

The main concepts of OZ are:

- *Schema*: It can been seen as a set [SIJ88].
- *Class*: It can been seen as a grouping of a *state schema*, *initial state schema* and *operation schemas* [TDC04]. It represents the object oriented approach

To illustrate those main concepts, consider the vending machine example denoted by $VM$:

- *Class*: To model the vending machine we need to define a class $VM$. Syntactically, in OZ a class definition is a named box as shown in Figure 2.7, where the dots ... refer to details explained next.

*VM*

$\ldots$

Figure 2.7: VM *class.*

- *State space*: The state space of our vending machine can be seen as a set of all valid states. The set of all valid states is:
  - In mathematics: $State\_Space = \{cv, tv : \mathbb{Z} \mid (0 \leq cv \leq 3) \wedge (0 \leq tv \leq 3) \bullet (cv, tv)\} = \{(0,0), \ldots, (3,3)\}$.
  - In OZ: The set $State\_Space$ can be described using a *state schema*, which is a box without name added to the class box as shown in Figure 2.8.

*VM*

$cv, tv : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

Figure 2.8: VM class: state schema.

- *Initial state*: Our vending machine has an initial state with $cv = 3$ and $tv = 3$. The set of all possible initial states, that respects those conditions is:
  - In mathematics: $Initial\_States = \{cv, tv : \mathbb{Z} \mid (0 \leq cv \leq 3) \wedge (0 \leq tv \leq 3) \wedge (cv = 3) \wedge (tv = 3) \bullet (cv, tv)\} = \{(3, 3)\}$.
  - In OZ: the set *Initial_States* can be described using a *initial state schema*, which is a box named *INIT* added to the class box as shown in Figure 2.9.

*VM*

$cv, tv : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

INIT

$cv = 3$
$tv = 3$

Figure 2.9: VM class: initial state schema.

- *State transition*: When the vending machine sells a coffee, the amount of coffee should be decreased by one. This is a state transition. The set of all possible state transitions when the selling coffee operation occurs is:
  - In mathematics: $coffee = \{cv, tv, cv', tv' : \mathbb{Z} \mid (0 \leq cv \leq 3) \wedge (0 \leq tv \leq 3) \wedge (0 \leq cv' \leq 3) \wedge (0 \leq tv' \leq 3) \wedge (tv' = tv) \wedge (cv' = cv - 1) \bullet ((cv, tv), (cv', tv'))\} = \{((3, 3), (2, 3)), \ldots, ((1, 0), (0, 0))\}$, where $(cv, tv)$

represents the *pre state* and $(cv', tv')$ represents the *post state* of a state transition.

– In OZ: the set *coffee* can be described using an *operation schema*, which is a box named with the operation name added to the class box as shown in Figure 2.10 left.

*VM*

$cv, tv : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

*Init*

$cv = 3$
$tv = 3$

*coffee*

$cv, tv, cv', tv' : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$
$0 \leq cv' \leq 3$
$0 \leq tv' \leq 3$
$tv' = tv$
$cv' = cv - 1$

*VM*

$cv, tv : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

*Init*

$cv = 3$
$tv = 3$

*coffee*

$\Delta(cv)$

$cv' = cv - 1$

*VM*

$cv, tv : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

*Init*

$cv = 3$
$tv = 3$

*coffee*

$\Delta(cv)$

$cv' = cv - 1$

*tea*

$\Delta(tv)$

$tv' = tv - 1$

Figure 2.10: VM class: operation schema.

OZ offers a more nice way to write the operation schema using $\Delta$-list. In OZ:

– Operation schema has a $\Delta$-list of state variables whose values may change. By convention, no $\Delta$-list means no attribute changes value.

– Operation schema implicitly includes the state schema and a primed version of it.

Thus, since the schema operation *coffee* specifies changes on the *coffee* value only, we can write it as shown in Figure 2.10 middle. Similarly, the operation schema *tea* is shown in Figure 2.10 right.

***Operation's input and output parameters*:**

Some operations can have input and output parameters, just like method in programming language, where the method's parameters represent the input, and the returned values represent the output. To illustrate the idea let us extend our vending machine .The new *VM* can talk to a shop sending a message to it. So it has a new operation *talk* and a state variable $m$ representing the message to be be sent.

The set of all possible state transitions when the *talk* operation occurs is:

- In mathematics: $talk = \{cv, tv, message, cv', tv', message', y : \mathbb{Z} \mid (0 \leq cv \leq 3) \wedge (0 \leq tv \leq 3) \wedge (0 \leq cv' \leq 3) \wedge (0 \leq tv' \leq 3) \wedge (tv' = tv) \wedge (cv' = cv) \wedge (message' = message) \wedge (y = message) \bullet ((cv, tv, message), (cv', tv', message'))\} = \{((3,3,1),(3,3,1)), \ldots, ((0,0,1),(0,0,1))\}$.

- In OZ: the set *talk* can be described using an *operation schema*, as shown in Figure 2.11. We can notice that this operation doesn't change any state variable's value, it just says that the value of the output parameter $y$, written as $y!$, must be equal to the value of the state variable *message*. For input parameter use ? symbol.

$$
\begin{array}{l}
\textbf{VM} \\
\hline
cv, tv, message : \mathbb{Z} \\
\hline
0 \leq cv \leq 3 \\
0 \leq tv \leq 3 \\
\hline
\textit{INIT} \\
cv = 3 \\
tv = 3 \\
message = 1 \\
\hline
\textit{coffee} \\
\Delta(cv) \\
\hline
cv' = cv - 1 \\
\hline
\textit{tea} \\
\Delta(tv) \\
\hline
tv' = tv - 1 \\
\hline
\textit{talk} \\
y! : \mathbb{Z} \\
\hline
y! = message
\end{array}
$$

Figure 2.11: VM class: *talk* operation with output parameter.

***Instance reference*:**

OZ is an object oriented approach, Thus every instance of a class needs a unique identifier, i.e., a reference name to refer to it. In OZ this can be seen simply as state constant *self* initialized with some *id* when the instance is created. Furthermore, operations can can share the instance identity through output or input the reference name *self* as shown in Figure 2.12 in the operation *talk*.

$VM(id : \mathbb{Z})$

$self, cv, tv, message : \mathbb{Z}$

$$0 \leq cv \leq 3$$
$$0 \leq tv \leq 3$$

$\textsc{Init}$

$$self = id$$
$$cv = 3$$
$$tv = 3$$
$$message = 1$$

$coffee$

$\Delta(cv)$

$$cv' = cv - 1$$

$tea$

$\Delta(tv)$

$$tv' = tv - 1$$

$talk$

$y! : \mathbb{Z}$
$z! : \mathbb{Z}$

$$y! = message$$
$$z! = self$$

Figure 2.12: VM class: instance reference.

### 2.2.2 Semantics

To understand the operational semantics of OZ we will use a labelled transition system LTS. Using this LTS we can investigate the state evolution of a OZ object . The definition of this LTS is adapted from Definition 2.1.5 with some changes.

**Definition 2.2.1 (LTS of OZ)** The labelled transition system $(\mathcal{S}^{OZ}, \mathcal{T})$ of OZ class states over the set of operations, has the valid states $\mathcal{S}^{OZ}$ as its states and its transitions $\mathcal{T}$ are those which can be inferred from the following rule:
$\underline{\textit{OPER}} : PRE\_STATE \xrightarrow{operation} POST\_STATE.$ $\triangle$

An example of using the transition rule of this LTS is: drawing the transition graph of vending machine shown in Figure 2.10. The transition graph is shown in Figure 2.13, where we show only a small part of it. The transitions *coffee* and *tea* refer to the operation schema *coffee* and *tea*. Thus, using the LTS we can enumerate all possible states transitions of an OZ state.

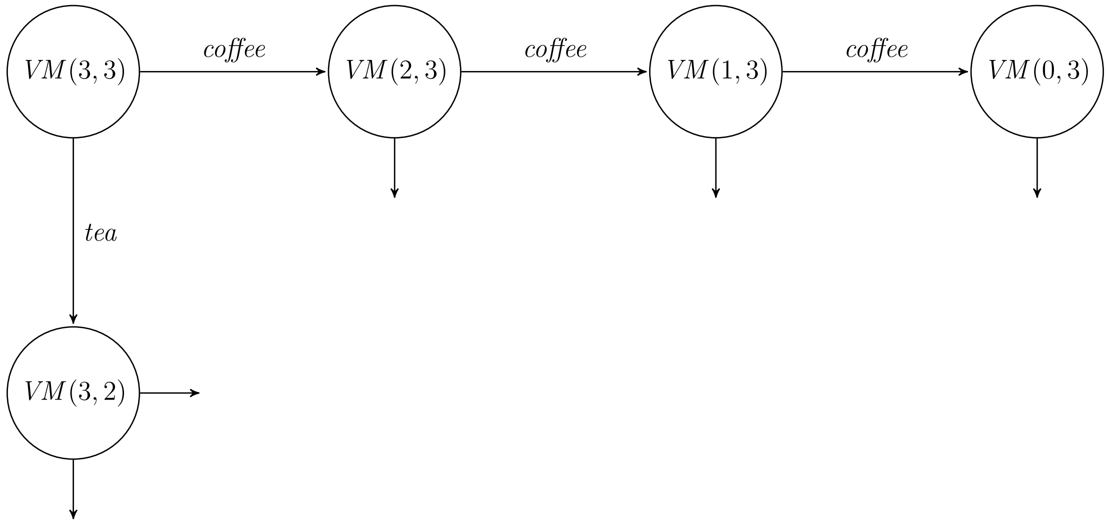



Figure 2.13: VM transition graph

### 2.2.3 Dynamic OZ

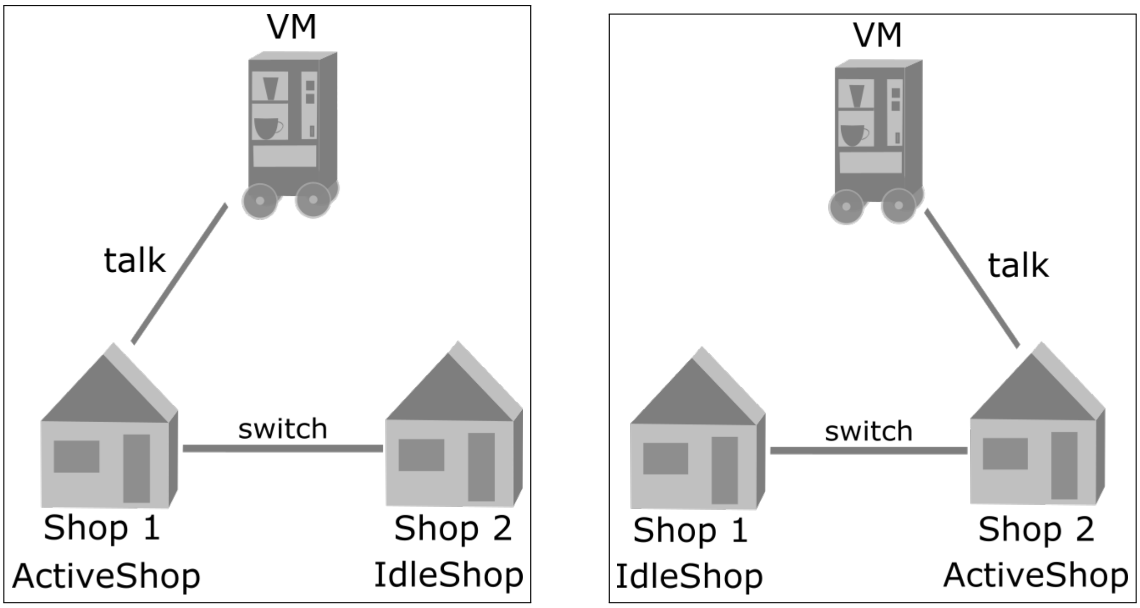
OZ can be used to model an entity with unchanged behavior, but sometimes we need to model an entity that changes its behavior. We introduce dynamic OZ, which is a version of OZ that uses the state pattern to model an entity with varying behavior.

#### OZ and state pattern

The state pattern is a behavioral software design pattern that allows an object to alter its behavior when its state changes. This pattern is close to the concept of finite-state machines. Changing behavior can be seen as that the object has changed its class. To illustrate the idea imagine that our vending machine *VM* is a mobile vending machine and that it is connected by a wireless link *talk* to a shop *Shop1*. On signal fading, *Shop1* decides to send the link *talk* to another shop *Shop2* through the link *switch* as shown in Figure 2.14. *Shop1* and *Shop2* change their behavior after switching. This varying behavior of shop can be handled through using two classes *ActiveShop* and *IdleShop* . A shop changes its class when *switch* occurs:

- *Shop1* sends *talk* via *switch* and changes its class from *ActiveShop* to *IdleShop* as shown in Figure 2.15.
- *Shop2* receives *talk* via *switch* and changes its class from *IdleShop* to *ActiveShop* as shown in Figure 2.15.

Notice, when an object changes its class it keeps its state variables and skips the *Init* schema of the new class. Dynamic OZ is based on the Agent-Place model used by MobileOZ described in [TDC02]. MobileOZ has two essential entities, agents and places. The main difference in the roles of these entities is that agents can move around the network, while places cannot. Dynamic OZ takes another approach by allowing places transferring, as shown in Figure 2.14, where the link *talk* can be transferred from *Shop*1 to *Shop*2. In Dynamic OZ, mobility is achieved by attaching a distinguished variable transferableOperation for storing names of locations. Location transferring is mimicked by assigning a new location to that variable.




(a) Before switching (b) After switching

Figure 2.14: Mobile vending machine and shops

$ActiveShop(id : \mathbb{Z})$

$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

$\textsc{Init}$
$self = id$
$transferableOperation = talk$

$switch$ $then$ $IdleShop$
$x! : nil \mid talk$
$x! = transferableOperation$
$transferableOperation' = nil$

$talk$
$\Delta(vmId, message)$
$y?, z? : \mathbb{Z}$
$y? = message'$
$z? = vmId'$

$IdleShop(id : \mathbb{Z})$

$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

$\textsc{Init}$
$self = id$
$transferableOperation = nil$

$switch$ $then$ $ActiveShop$
$\Delta(transferableOperation)$
$x? : nil \mid talk$
$x? = transferableOperation'$

Figure 2.15: active and idle shop

### Restriction

In this work when we use OZ to model an operation, we restrict our self to use only one type of parameters in the operation schema. Either input or output. This can be noticed in the operation schema *talk* in:

- In Figure 2.12 all the parameters of the operation schema *talk* are output paramaters.
- In Figure 2.15 all the parameters of the operation schema *talk* are input paramaters.

Why this restriction? Because a channel in $\pi$-calculus is unidirectional pro reaction. In the next chapter we will map the OZ class constructs to $\pi$-calculus constructs, so we will map an OZ operation to an $\pi$-calculus name, i.e., channel. In $\pi$-calculus a processes can send or receive over a channel pro reaction, but not the both together.

# 3 Transformational semantics of OZ

This chapter studies the syntactic transformation of an OZ class into an $\pi$-calculus process. The resulting process is intuitively defined as follow:
$P_{OZ\_PI} = \sum_{v\_st \in Init} \tau . Q(v\_st, v\_self)$ with
$Q(v\_st, v\_self) = (\sum_{c \in \mathrm{In}} c(v\_in_c) + \sum_{c \in \mathrm{Out}} \tau . c < v\_out_c >) \ . \ \sum_{v\_st'} \tau . Q(v\_st', v\_self)$
where:

- `In` the set of all *input actions*, defined as `In` $=_{\text{def}} \{x(\vec{y}) \ \mid \ x \in \mathcal{N}\}$.
- `Out` is the set of all *output actions*, defined as `Out` $=_{\text{def}} \{\overline{x}\langle\vec{y}\rangle \ \mid \ x \in \mathcal{N}\}$.
- $c$ refers to an operation.
- $v\_st$ refers to the variables of the current state.
- $v\_st'$ refers to the variables of the successor state.
- $v\_self$ refers to the instance reference.
- $c.v\_in_c$ refers to the occurrence of the operation $c$, where $v\_in_c$ represents the values of the input parameters of $c$.
- $c.v\_out_c$ refers to the occurrence of the operation $c$, where $v\_out_c$ represents the values of the output parameters of $c$.

What is the benefit of transforming an OZ class into $\pi$-calculus process? The main advantage is that it can be combined, using the parallel operator, with a second, explicit pi-calculus process that represents the desired sequencing of the operations of the OZ class. This will enable us to study the behavior of an entity as will be shown later in the next chapter.

To transform an OZ class into a $\pi$-calculus process we need to remember that the $\pi$-calculus has only names and processes, and nothing else. A *name* in $\pi$-calculus can be seen as a *channel* or a *memory location*. Thus, in this work when we use the word *channel* we refer to a $\pi$-calculus *name*. We need to use the names and processes to represent: value, state variable, state schema, initial state schema and operation schema in $\pi$-calculus.

## 3.1 Mapping values

We consider a finite set of natural numbers represented as binary numbers shown Table 3.1. A value can be mapped to a $\pi$-calculus process. Listing 3.1 shows the $\pi$-calculus implementation of the values 0,1,2,3 in ABC syntax. The keyword *agent* defines a new processes. The process *Zero* is modeled using alternative choice: it either receives a signal via the channel *a* and switch off, or it receives two channels *tt,ff* via *a*, then it sends two signals via the channel *ff*.

| Decimal | Binary |
|---|---|
| 0 | 00 |
| 1 | 01 |
| 2 | 10 |
| 3 | 11 |

Table 3.1: Two bits binary numbers.

```
agent Zero(a) = a(tt,ff).'ff.'ff.Zero(a) + a.0
agent One(a) = a(tt,ff).'tt.'ff.One(a) + a.0
agent Two(a) = a(tt,ff).'ff.'tt.Two(a) + a.0
agent Three(a) = a(tt,ff).'tt.'tt.Three(a) + a.0
```

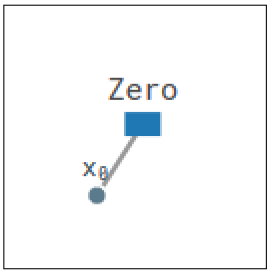
Listing 3.1: 0,1,2,3 as $\pi$-calculus processes.

## 3.2 Mapping state variables

A variable can be mapped to a channel. Creating a variable $x$ and initializing it with the value 0 ( int x = 0;) is mapped to creating a new channel x and initialize the processes Zero with the channel x as shown in Figure 3.1. The wide hat refers to creating a new channel.

(^x ) ( Zero(x) )

(a) ABC code.




(b) visualization.

Figure 3.1: variable as a channel

Thus, we map the state variables *self*, *cv*, *tv*, *message* of Figure 2.12 to $\pi$-calculus channels *self*, *cv*, *tv*, *message* as shown in Figure 3.2

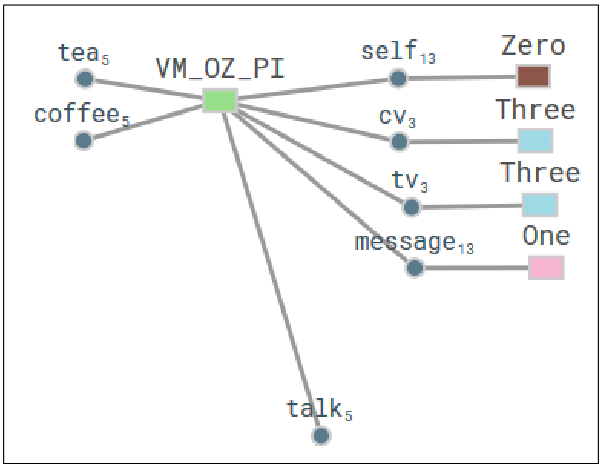



Figure 3.2: VM as a $\pi$-calculus process VM_OZ

## 3.3 Mapping operations

We map OZ class operations to $\pi$-calculus channels as we did with state variables. That is, we map the operations *coffee*, *tea*, *talk* of Figure 2.12 to $\pi$-calculus channels *coffee*, *tea*, *talk* as shown in Figure 3.2

## 3.4 Mapping data Types

For simplicity, we don't implement any kind of type checking, but we deal with types by representing the value of the type by corresponding process. For example $cv : \{0, 1, 2, 3\}$, means that the allowed processes to be initialized with $cv$ are: *Zero*, *One*, *Two*, *Three*.

## 3.5 Mapping mathematical operators

***Addition***:

To add two numbers we need an addition processes that mimics the behavior of arithmetic circuits for adding two bits binary numbers shown in Figure 3.3. Figure 3.4 shows visualization of the addition process and the ABC code. The full implementation of *Add* processes can be found in the appendix.

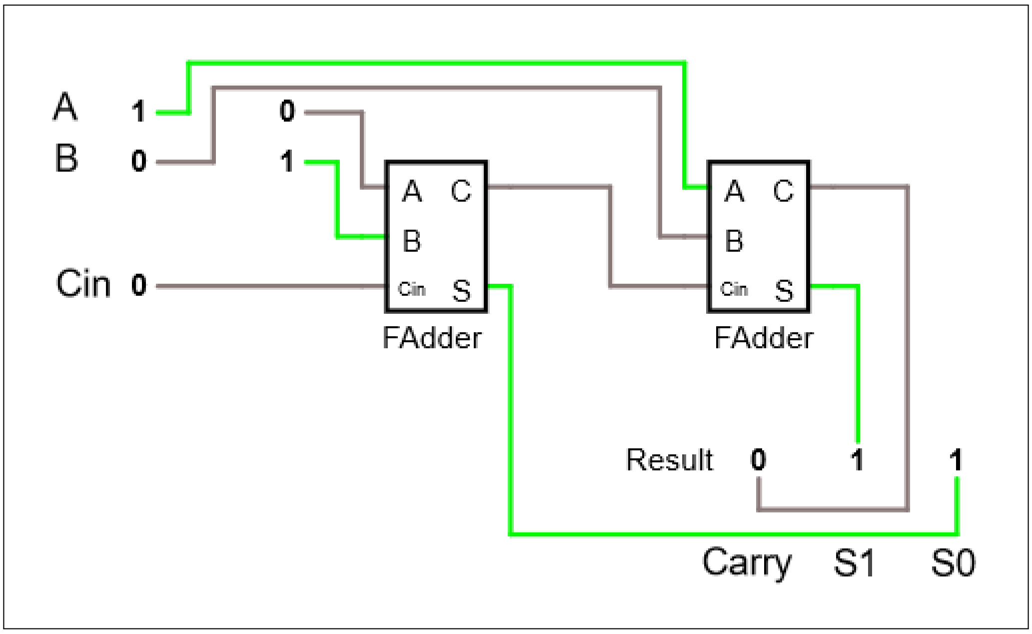
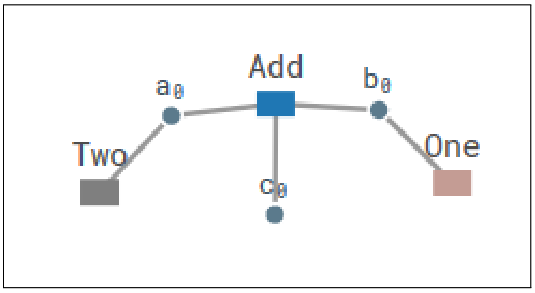
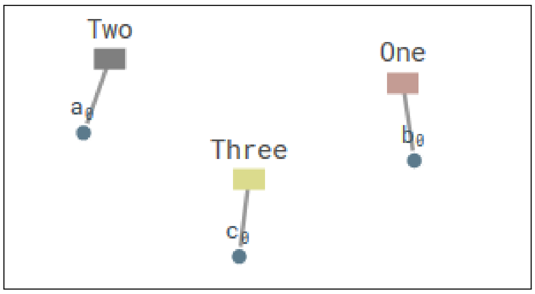



Figure 3.3: adder circuit




(a) Before addition.




(b) After addition.

| (^a,b,c ) ( Two(a) \| One(b) \| Add(a,b,c) ) |
|---|

(c) Abc code.

Figure 3.4: addition as a process

***Subtraction*:**

To subtract two numbers we use an subtraction process that mimics the behavior of arithmetic circuits for subtracting two bits binary numbers shown in Figure 3.5. Figure 3.6 shows visualization of the subtraction process and the ABC code. The full implementation of *Sub* processes can be found in the appendix.

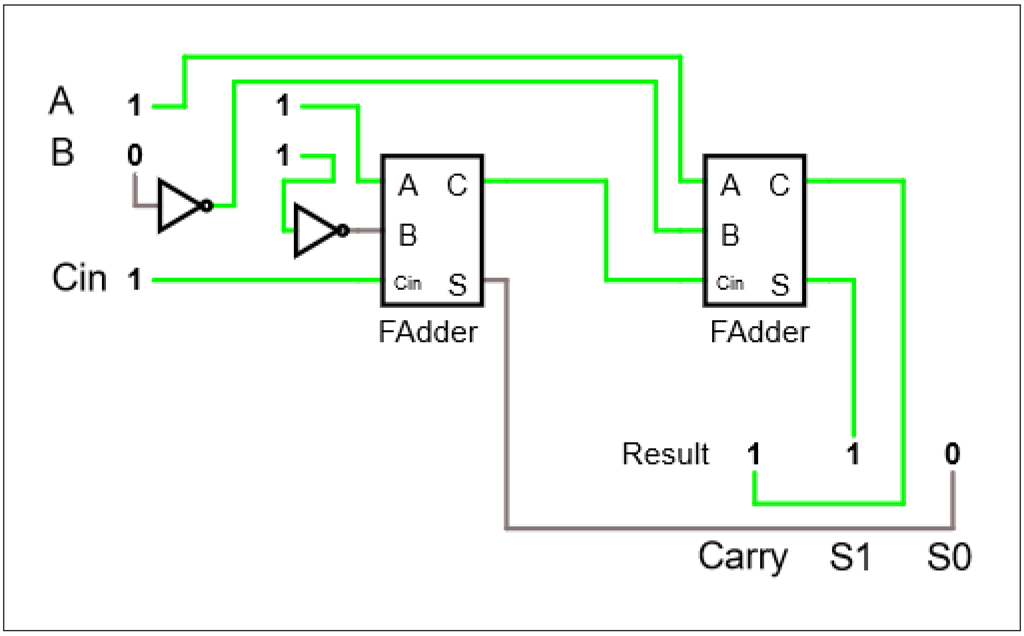
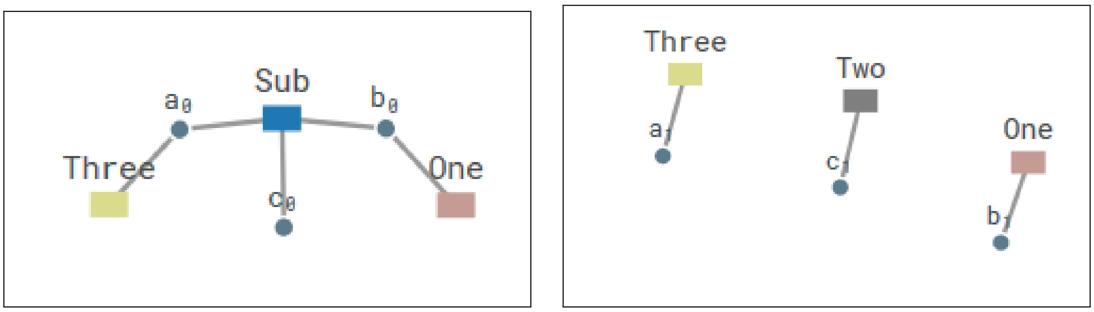



Figure 3.5: subtractor circuit




(a) Before subtraction.

(b) After subtraction.

( ^a,b,c ) ( Three(a) | One(b) | Sub(a,b,c) )

(c) ABC code.

Figure 3.6: subtraction as a process

***Comparation*:**

To compare two numbers we use a process that mimics the behavior of arithmetic circuits for comparing two bits binary numbers shown in Figure 3.7. Figure 3.8 shows visualization and ABC code of the comparator process and a simple if-else statement. The full implementation of *Compare* processes can be found in the appendix.

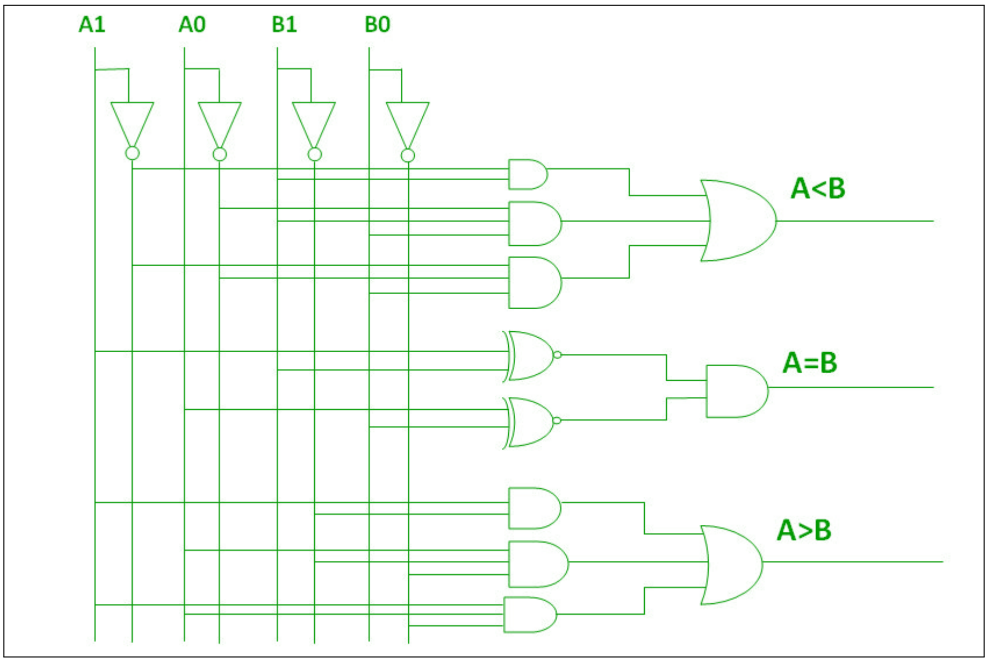
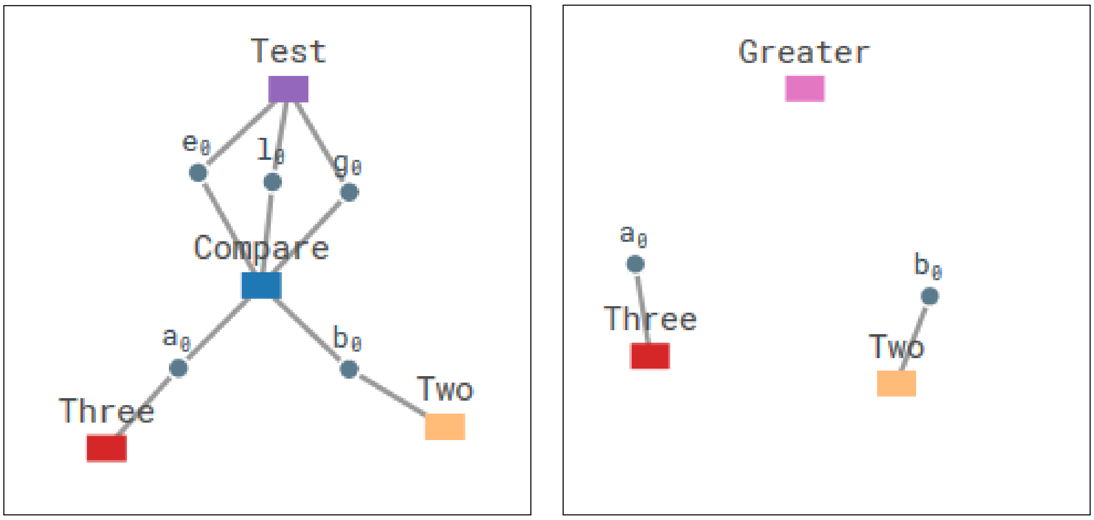



Figure 3.7: comparator circuit




(a) Before comparation. (b) After comparation.

| (^a,b,g,e,l ) ( Three(a) \| Two(b) \| Compare(a,b,g,e,l) \| If_Else(g,e,l) ) |
|---|

(c) ABC code.

Figure 3.8: comparation as a process

***Set union and subtraction***:

The implementation of set union and abstraction processes can be found in the appendix.

## 3.6 Mapping OZ class

The class *VM* shown in Figure **??** is mapped to a $\pi$-calculus process *VM_OZ_PI* shown in Listing 3.2. The processes **VM_OZ** has six parameters

- *self,message,cv* and *tv* represent the state variables.
- *coffee,tea* and *talk* represent the operations.

The processes *VM_OZ_PI* mimics the behaviour of *VM*:

- On receiving a signal via *coffee*, then *VM_Condition_IF_Else_coffee* checks if the condition *VM_Condition_coffee* is fulfilled. If it is fulfilled it makes a state transition *VM_State_Transition_coffee* to decreases the value of *cv* by one *One(b)* | *Sub(cv,b,c,done)*.

- the same goes for *tea*.
- *VM_OZ_PI* can send a copy of the value of *self,message* via *talk*

The processes *VM_OZ_PI_Init* creates an instance of *VM_OZ_PI* and initialize its state variables *self,cv,tv* and *message* with the values *Zero,Three,Three,One*. The full implementation can be found in the appendix.

$VM(id : \mathbb{Z})$

$self, cv, tv, message : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

INIT

$self = id$
$cv = 3$
$tv = 3$
$message = 1$

*coffee*

$\Delta(cv)$

$cv' = cv - 1$

*tea*

$\Delta(tv)$

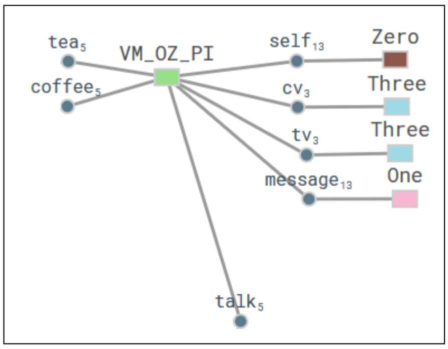
$tv' = tv - 1$

*talk*

$y! : \mathbb{Z}$
$z! : \mathbb{Z}$

$y! = message$
$z! = self$

(a) VM class in OZ




(b) stargazer visualization

Figure 3.9: transforming VM into $\pi$-calculus process VM_OZ_PI

```
agent VM_OZ_PI_Init(coffee,tea,talk) = (^self,cv,tv,message)
    (VM_OZ_PI(self,coffee,tea,talk,cv,tv,message) | Zero(
    self) | Three(cv) | Three(tv) | One(message))

agent VM_OZ_PI(self,coffee,tea,talk,cv,tv,message) =
coffee.(^res_t,res_f) (VM_Condition_coffee(self,coffee,tea,
    talk,cv,tv,message,res_t,res_f) |
    VM_Condition_IF_Else_coffee(self,coffee,tea,talk,cv,tv,
    message,res_t,res_f)) \
+ tea.(^res_t,res_f) (VM_Condition_tea(self,coffee,tea,talk,
    cv,tv,message,res_t,res_f) | VM_Condition_IF_Else_tea(
    self,coffee,tea,talk,cv,tv,message,res_t,res_f)) \
+ (^m_c,m_done,r_c,r_done) ( (m_done.r_done.'talk<r_c,m_c>.
   VM_OZ_PI(self,coffee,tea,talk,cv,tv,message)) | Copy(
    message,m_c,m_done) | Copy(self,r_c,r_done))

agent VM_Condition_coffee(self,coffee,tea,talk,cv,tv,message
    ,res_t,res_f) = (^b,g,e,l) (Zero(b) | Compare(cv,b,g,e,l)
     | CleanAndTF(g,e,l,res_t,res_f,b))
agent VM_Condition_IF_Else_coffee(self,coffee,tea,talk,cv,tv
    ,message,res_t,res_f) = res_t.(VM_State_Transition_coffee
    (self,coffee,tea,talk,cv,tv,message)) + res_f.
    VM_PleaseFillMe
agent VM_State_Transition_coffee(self,coffee,tea,talk,cv,tv,
    message) = (^sub_done, b,c,done) ((One(b) | Sub(cv,b,c,
    done) | ClearThenCopy(cv,b,c,done,sub_done)) | (sub_done
    .'coffee.VM_OZ_PI(self,coffee,tea,talk,cv,tv,message)) )
```

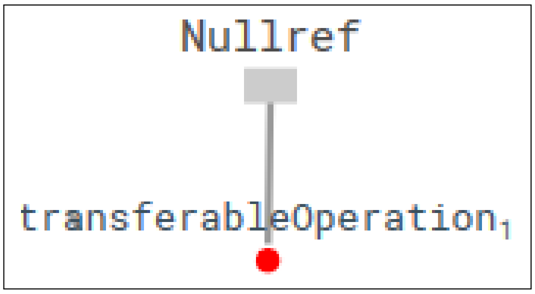
Listing 3.2: the process VM_OZ_PI in ABC code.

## 3.7 Mapping transferable operation's variable

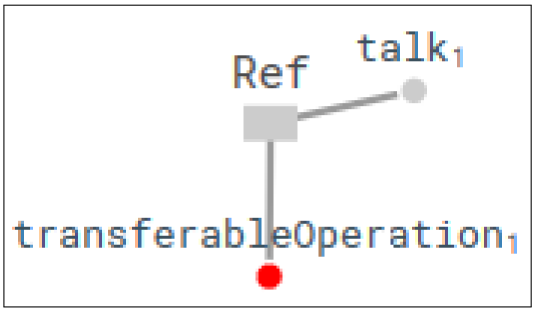
As mentioned in Section 2.2.3, the mobility in Dynamic OZ is achieved by attaching a distinguished variable transferableOperation for location. Location transferring is mimicked by assigning a new location to that variable. To translate the variable transferableOperation into $\pi$-calculus we cannot use $\pi$-calculus channel as in Section 3.2, since the value of transferableOperation will be a channel name and not a processes representing a value like *Zero*. Thus, we map the variable transferableOperation to a channel named transferableOperation, where:

- transferableOperation = nil is mapped to Nullref(transferableOperation),
- transferableOperation = talk is mapped to Ref(transferableOperation,talk),

as shown in Figure 3.10 and Listing 3.3.




(a) transferableOperation = nil.




(b) transferableOperation = talk

Figure 3.10: mapping transferable operation's variable

```
agent Nullref(r) = r(n,c).('n.Nullref(r) + c(m).Ref(r,m) + n
    .Nullref(r))
agent Ref(r,v) = r(n,c).('c<v>.Ref(r,v) + c(m).Ref(r,m) + 'n
    .Nullref(r))
```

Listing 3.3: Nullref and Ref processes in ABC code.

Thus using the concept of transferable operation's variable we can now transform the classes *IdleShop* and *ActiveShop* as shown in Figure 3.11, Listing 3.4 and Figure 3.12, Listing 3.5.

*IdleShop*$(id : \mathbb{Z})$

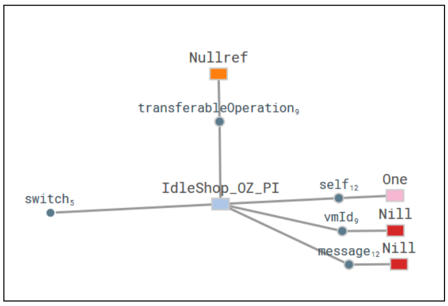
$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

INIT
$self = id$
$transferableOperation = nil$

*switch* *then ActiveShop*
$\Delta(transferableOperation)$
$x? : nil \mid talk$

$x? = transferableOperation'$

(a) IdleShop class in OZ




(b) stargazer visualization

Figure 3.11: transforming IdleShop into $\pi$-calculus process IdleShop_OZ_PI

```
agent IdleShop_OZ_PI(self,switch,transferableOperation,vmId,
    message) = switch(talk_new).((^n,c) ('
    transferableOperation<n,c>.'c<talk_new>.ActiveShop_OZ(
    self,switch,transferableOperation,talk_new,vmId,message))
    )

agent IdleShop_OZ_PI_Init_Null(switch) = (^self,
    transferableOperation,vmId,message) (IdleShop_OZ_PI(self,
    switch,transferableOperation,vmId,message) | One(self) |
    Nullref(transferableOperation) | Nill(vmId) | Nill(
    message))
```

Listing 3.4: the process IdleShop_OZ_PI in ABC code.

$ActiveShop(id : \mathbb{Z})$

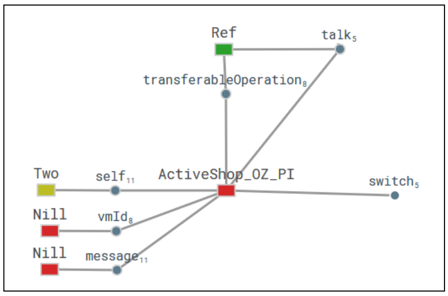
$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

$\textsc{Init}$
$self = id$
$transferableOperation = talk$

$switch$ then $IdleShop$
$x! : nil \mid talk$
$x! = transferableOperation$
$transferableOperation' = nil$

$talk$
$\Delta(vmId, message)$
$y?, z? : \mathbb{Z}$
$y? = message'$
$z? = vmId'$

(a) ActiveShop class in OZ




(b) stargazer visualization

Figure 3.12: transforming ActiveShop into $\pi$-calculus process ActiveShop_OZ_PI

```
agent ActiveShop_OZ_PI(self,switch,transferableOperation,
    talk_current,vmId,message) =
'switch<talk_current>.( ((^n,c) ('transferableOperation<n,c
    >.'n.IdleShop_OZ(self,switch,transferableOperation,vmId,
    message))) | KillAndSetNilIfNotNill(vmId)   |
    KillAndSetNilIfNotNill(message))   \
+ talk_current(vmId_new,m_new).(^done_ref,done_m) ( (
    done_ref.done_m.ActiveShop_OZ_PI(self,switch,
    transferableOperation,talk_current,vmId,message))  |
    KillThenCopyValueThenKillTemp(vmId_new,vmId,done_ref)  |
    KillThenCopyValueThenKillTemp(m_new,message,done_m))

agent ActiveShop_OZ_PI_Init_Talk(switch,talk_current) = (^
    self,transferableOperation,vmId,message) (
```

```
ActiveShop_OZ_PI(self, switch, transferableOperation,
talk_current, vmId, message) | Two(self) | Ref(
transferableOperation, talk_current) | Nill(vmId) | Nill(
message))
```

Listing 3.5: the process ActiveShop_OZ_PI in ABC code.

Finally, Figure 3.13, Figure 3.14 and Listing 3.6 show the big picture of a system consisting of two shops, a vending machine and a customer. The full implementation can be found in the appendix.

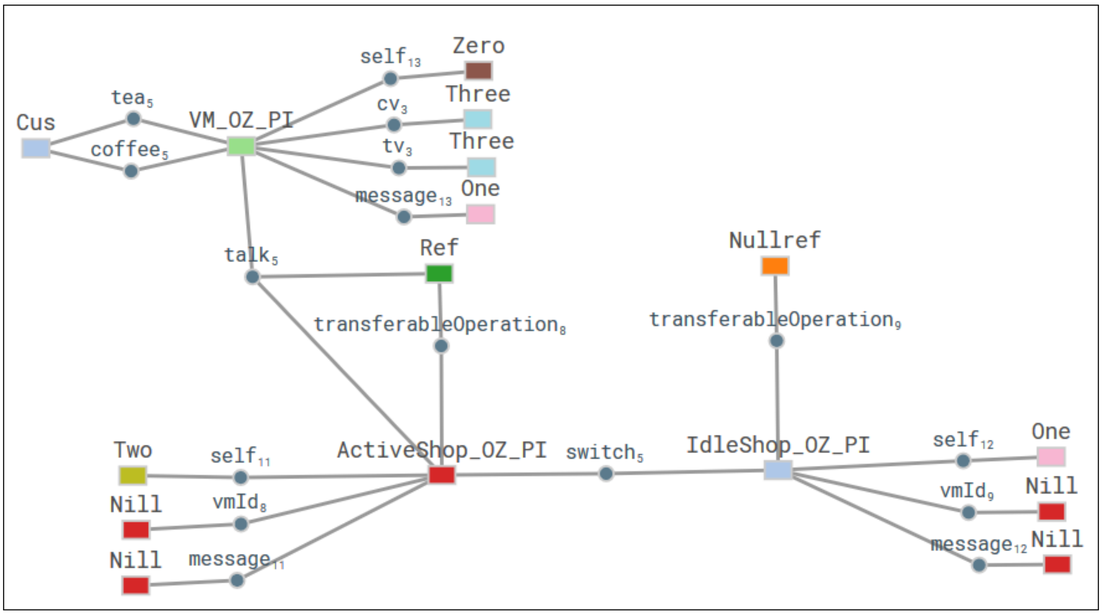



Figure 3.13: System before switching

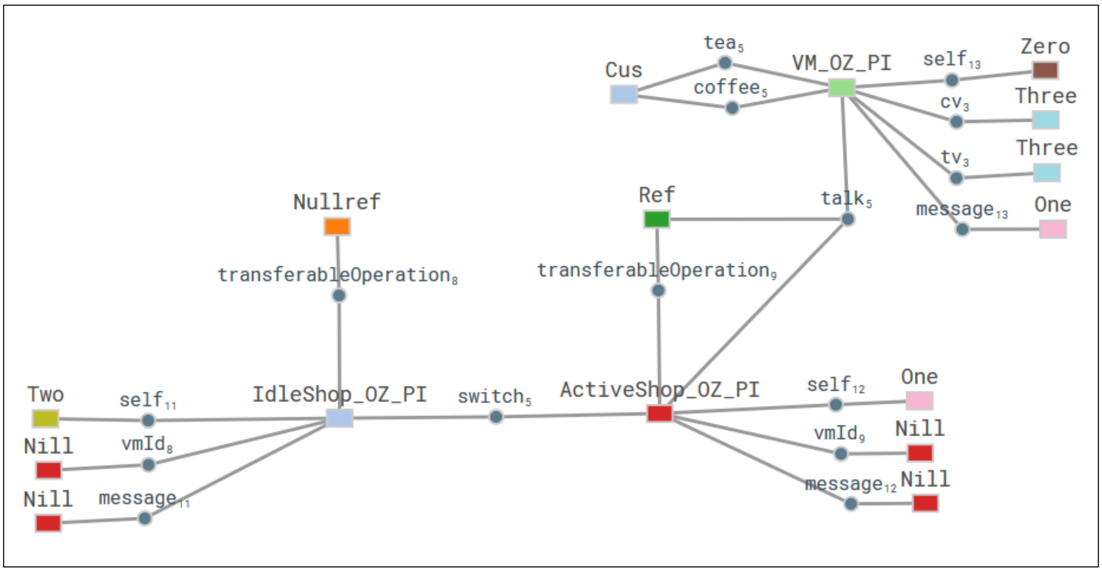



Figure 3.14: System after switching

```
agent System = (^coffee, tea, switch, talk) ( VM_OZ_PI_Init(
    coffee, tea, talk) | Cus(coffee, tea) |
    IdleShop_OZ_Init_Null(switch) |
    ActiveShop_OZ_PI_Init_Talk(switch, talk))
```

Listing 3.6: the system consisting of: two shops, vending machine and a customer in ABC code.

# 4 The combination $\pi$-OZ

In this chapter we will combine the specifications languages OZ and $\pi$-calculus into the combination $\pi$-OZ, and we will study its transformational semantics.

## 4.1 Syntax

Syntactically the $\pi$-OZ specification is divided into an interface, a $\pi$ part and an OZ part as shown in Figure 4.1. The idea of the combination is that communication in the $\pi$ part has effects on the state space of the OZ part as specified in its operation schema. Figure 4.2 shows the $\pi$-OZ specification of our vending machine $VM$. In the interface, all channels are declared with the associated types. The $\pi$ part is a system of recursive equations written according to the $\pi$-calculus syntax represents the sequencing of operations as shown in $VM\_PI$. In the OZ part the state space, the initial schema and the operation schemes introduced, where the operation schemes defines the effect of communications on the in specified channels.

$S$

$Interface$
$S_{\pi\_part}$
$S_{OZ\_part}$

Figure 4.1: $\pi$-OZ specification of an entity $S$

$VM(id : \mathbb{Z})$

$$chan\ coffee, tea$$
$$chan\ talk : \mathbb{Z} \times \mathbb{Z}$$

$$\begin{aligned} VM\_PI = {} & coffee().VM\_PI + tea().VM\_PI \\ & + talk < self, message > .VM\_PI \end{aligned}$$

$self, cv, tv, message : \mathbb{Z}$

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

INIT

$self = id$
$cv = 3$
$tv = 3$
$message = 1$

*coffee*

$\Delta(cv)$

$cv' = cv - 1$

*tea*

$\Delta(tv)$

$tv' = tv - 1$

*talk*

$y! : \mathbb{Z}$
$z! : \mathbb{Z}$

$y! = message$
$z! = self$

Figure 4.2: π-OZ specification of the $VM$.

Figure 4.3 shows the π-OZ specification of the active and idle shop.

$ActiveShop(id : \mathbb{Z})$

$chan\ switch : nil \mid talk$
$chan\ talk : \mathbb{Z} \times \mathbb{Z}$

$ActiveShop\_PI =$
$talk(vmId, message).ActiveShop\_PI$
$+switch < talk > .IdleShop\_PI$

$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

INIT
$self = id$
$transferableOperation = talk$

$switch\_\_\ then\ IdleShop$
$x! : nil \mid talk$
$x! = transferableOperation$
$transferableOperation' = nil$

$talk$
$\Delta(vmId, message)$
$y?, z? : \mathbb{Z}$
$y? = message'$
$z? = vmId'$

$IdleShop(id : \mathbb{Z})$

$chan\ switch : nil \mid talk$

$IdleShop\_PI =$
$switch(transferableOperation).ActiveShop\_PI$

$self, vmId, message : \mathbb{Z}$
$transferableOperation : nil \mid talk$

INIT
$self = id$
$transferableOperation = nil$

$switch\_\_\ then\ ActiveShop$
$\Delta(transferableOperation)$
$x? : nil \mid talk$
$x? = transferableOperation'$

Figure 4.3: $\pi$-OZ specification of the active and idle shop

## 4.2 Transformational semantics

The semantics of the combination $\pi$-OZ specification S can then be described by the $\pi$-calculus process $S_{OZ\_part_\pi} \mid S_{\pi\_part}$, where $S_{OZ\_part_\pi}$ is the syntactic transformation of OZ part into $\pi$-calculus process. For example, the semantics of Figure 4.2 is $VM\_OZ\_PI \mid VM\_PI$, where $VM\_OZ\_PI$ is as descriped in Listing 3.2. Unfortunately, this will not work well, since the parallel operator $\mid$ only allows the binary synchronization via a channel, not like in *CSP* where the parallel operator $||$ allows the multiple synchronization via a channel. That will be problematic when we try to combine the $\pi$-OZ specification of an entity S with a $\pi$-OZ specification of another entity R in parallel. To solve this problem we can use **broadcast channel** or **non-atomic reaction** concept as follow:

### *Shared Channel:*

To allow the multiple synchronization via a channel in $\pi$-calculus, we use concept of the shared channel. [En99] introduces the b$\pi$, which is an extension of $\pi$-calculus implementing broadcast communications. Additionaly, the UPPAAL model checker introduces the broadcast channel too [Ol08]. For simplicity, we use the broadcast channel from UPPAAL with a little change. On a shared channel one sender synchronizes with an at least one receiver. Thus, like binary synchronization, a shared channel blocks the sender if there are no receivers. Furthermore, we can send and receive on a shared channel. We extend the transition rules of $\pi$-calculus defined in Definition 2.1.5, with an additional rule:

$$\underline{\mathit{Shared\_Chan\_PAR}} : \frac{P \xrightarrow{\overline{x}\langle\vec{y}\rangle} P' \quad Q \xrightarrow{x(\vec{y})} Q' \quad R \xrightarrow{x(\vec{z})} R'}{P \mid Q \mid R \xrightarrow{\tau} P' \mid \{\vec{y}/\vec{z}\}\, Q' \mid \{\vec{y}/\vec{z}\}\, R'} \quad \mathit{if}\ x : \mathit{Shared}$$

Figure 4.4: transition rule for shared channel.

Figure 4.5 shows the $\pi$-OZ specification of our $VM$ using shared channels. The combination's process $S_{OZ\_part_\pi} \mid S_{\pi\_part}$ for $VM$ is $VM\_OZ\_PI \mid VM\_PI$. The main advantage of the shared channels in $VM$ is that, if we combine the combination's processes with a third processes $Cus$ representing a customer which issues a signal on the *coffee* channel, this will enforce both $VM\_OZ\_PI$ and $VM\_PI$ to evolve, since they are listening on coffee, which is shared channel in $Cus$, $VM\_OZ\_PI$ and $VM\_PI$. The behavior of $VM$ can be seen as the intersection of the behavior of $VM\_OZ\_PI$ and $VM\_PI$ .i.e. the intersection of the transition graphs .i.e the automates. Unfortunately, our tools do not support the shared channel, thus we will not proceed with this approach.

$VM(id : \mathbb{Z})$

$shared\ chan\ coffee, tea$
$shared\ chan\ talk : \mathbb{Z} \times \mathbb{Z}$

$$VM\_PI = coffee().VM\_PI + tea().VM\_PI$$
$$+talk < self, message > .VM\_PI$$

| |
|---|
| $self, cv, tv, message : \mathbb{Z}$ |
| $0 \leq cv \leq 3$<br>$0 \leq tv \leq 3$ |

| INIT |
|---|
| $self = id$<br>$cv = 3$<br>$tv = 3$<br>$message = 1$ |

| $coffee$ |
|---|
| $\Delta(cv)$ |
| $cv' = cv - 1$ |

| $tea$ |
|---|
| $\Delta(tv)$ |
| $tv' = tv - 1$ |

| $talk$ |
|---|
| $y! : \mathbb{Z}$<br>$z! : \mathbb{Z}$ |
| $y! = message$<br>$z! = self$ |

Figure 4.5: $\pi$-OZ specification of the $VM$ using broadcast channels.

***Non-atomic reaction*:**

Let us examine the process $Cus \mid VM\_OZ\_PI \mid VM\_PI$ shown in Figure 4.6. When $Cus$ issues a signal on the *coffee* channel, it is required that $VM\_OZ\_PI$ and $VM\_PI$ receives the signal and evolve together. This is not possible, since the $\pi$-calculus communications are binary, so either $VM\_OZ\_PI$ or $VM\_PI$ will evolve and the other will not. To solve this problem using binary communications we propose to break the channel *coffee* down into two channels: $ex\_cofee$ and $in\_coffee$ as shown in Figure 4.7. The channel $ex\_cofee$ is for the external, outside $VM$, communication between $Cus$ and $VM\_PI$. The channel $in\_coffee$ is for the internal, inside $VM$, communication between $VM\_PI$ and $VM\_OZ\_PI$. In Figure 4.7 the numbered arrows represent the communication flow from $VM\_PI$'s point of view. $VM\_PI$ receives a signal via $ex\_coffee$ and re-sends it via $in\_coffee$. When $VM\_OZ\_PI$ ends its processing it sends a done signal via $done\_in\_coffee$ to $VM\_PI$ which re-sends the done signal to $Cus$ via $done\_ex\_coffee$. This way the combination's process $S_{OZ\_part_{\pi}} \mid S_{\pi\_part}$,i.e. $VM\_OZ\_PI \mid VM\_PI$, behaves as a one processes from the view point of its environment .i.e. $Cus$, by breaking down the channel, reproducing the signal, and using the done signal. All that makes the reaction *ordering a coffee* a non-atomic reaction. Furthermore, we can notice that the non-atomic reaction concept is overburdening, since we now have four channels *ex_coffee, in_coffee, done_in_coffee, done_ex_coffee* instead of having one channel for *coffee*.

Figure 4.8 shows how the $\pi$-OZ specification of $VM$ implements the non-atomic reaction concept. In the interface part it defines the needed channels. For *coffee* four channels: one external, one internal and two for done signaling. The internal channels $in\_coffee, done\_in\_coffee$ are invisible outside $VM$ Thus we need to extend OZ with a new construct *chan local* to define local channels. The local channel is like the *new* operator in $\pi$-calculus,i.e. restriction as follow $VM = new\ in\_coffee, done\_in\_coffee...(VM\_PI \mid VM\_OZ\_PI)$. For *tea* and *talk* the same is done like *coffee*. The behavior sequence part $VM\_PI = ex\_coffee().in\_coffee <> .done\_in\_coffee()....$ reflects exactly the numbered arrows shown in Figure 4.7.

The $\pi$-OZ specification of $VM$ reads: the combination is ready to participate in an $ex\_cofee$ action issued by the environment. On receiving a signal via $ex\_cofee$, the $\pi$ part will make a transition and issue a signal via $in\_coffee$ enforcing the OZ part to make a transition specified with the operation schema $in\_coffee$. When the OZ part ends its transition it sends a signal via $done\_in\_cofee$ enforcing the $\pi$ part to make a transition, and finally the $\pi$ part issues a done signal via $done\_ex\_cofee$ to the environment declaring that ordering a coffee has done successfully. Notice that the specification has a schema for $in\_coffee$ which represents the conditions on the data, and there no schemes for $ex\_cofee$, $done\_ex\_cofee$ and $done\_in\_cofee$, since they serve for orchestrating.

Figure 4.9 and Figure 4.10 show the $\pi$-OZ specification of *ActiveShop* and *IdleShop* respectively, using the non-atomic reaction concept. Figure 4.11 shows a big picture of a system consisting of a customer, vending machine and two shops

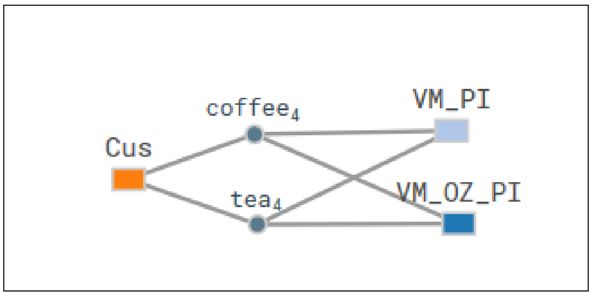



Figure 4.6: the process $Cus \mid VM\_OZ\_PI \mid VM\_PI$

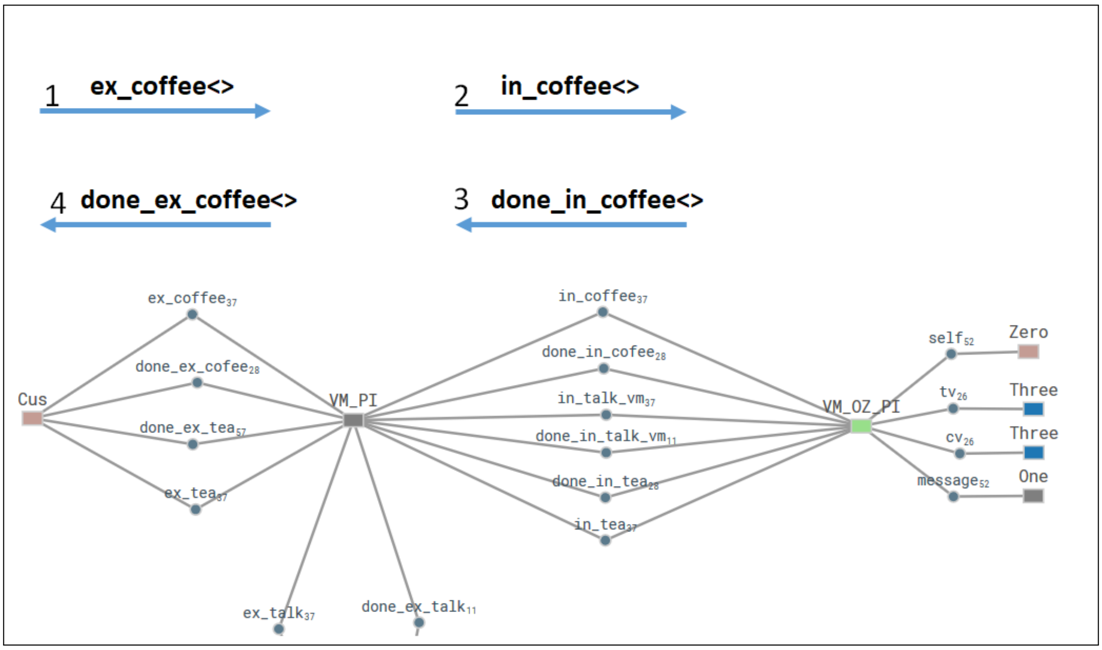



Figure 4.7: Action reproducing and non-atomic reaction

$VM(id : \mathbb{Z})$

$chan\ ex\_coffee, done\_ex\_coffee, ex\_tea, done\_ex\_tea$
$chan\ local\ in\_coffee, done\_in\_coffee, in\_tea, done\_in\_tea$
$chan\ ex\_talk : \mathbb{Z} \times \mathbb{Z}$
$chan\ local\ in\_talk : \mathbb{Z} \times \mathbb{Z}$
$chan\ done\_ex\_talk$
$chan\ local\ done\_in\_talk$

$$
\begin{aligned}
VM\_PI = {} & ex\_coffee().in\_coffee <> \\
& \quad .done\_in\_coffee().done\_ex\_coffee <> .VM\_PI \\
& + ex\_tea().in\_tea <> \\
& \quad .done\_in\_tea().done\_ex\_tea <> .VM\_PI \\
& + in\_talk(y, z).ex\_talk < y, z > \\
& \quad .done\_ex\_talk().done\_in\_talk <> .VM\_PI
\end{aligned}
$$

$self, cv, tv, message : \mathbb{Z}$

---

$0 \leq cv \leq 3$
$0 \leq tv \leq 3$

$\textsc{Init}$

$self = id$
$cv = 3$
$tv = 3$
$message = 1$

$in\_coffee$

$\Delta(cv)$

---

$cv' = cv - 1$

$in\_tea$

$\Delta(tv)$

---

$tv' = tv - 1$

$in\_talk$

$y! : \mathbb{Z}$
$z! : \mathbb{Z}$

---

$y! = message$
$z! = self$

Figure 4.8: $\pi$-OZ specification of the $VM$ using non-atomic reaction.

$ActiveShop(id : \mathbb{Z})$

$$
\begin{array}{l}
chan\ ex\_switch : nil \mid in\_talk \\
chan\ local\ in\_switch : nil \mid in\_talk \\
chan\ ex\_talk : \mathbb{Z} \times \mathbb{Z} \\
chan\ local\ in\_talk : \mathbb{Z} \times \mathbb{Z} \\
chan\ done\_ex\_talk \\
chan\ local\ done\_in\_talk
\end{array}
$$

$$
\begin{array}{l}
ActiveShop\_PI = \\
\quad ex\_talk(y, z).in\_talk < y, z > \\
\qquad\qquad .done\_in\_talk().done\_ex\_talk <> .ActiveShop\_PI \\
\quad + in\_switch(x).ex\_switch < x, ex\_talk > .IdleShop\_PI
\end{array}
$$

$$
\begin{array}{l}
self, vmId, message : \mathbb{Z} \\
transferableOperation : nil \mid in\_talk
\end{array}
$$

$\textsc{Init}$

$$
\begin{array}{l}
self = id \\
transferableOperation = in\_talk
\end{array}
$$

$in\_switch$ $then\ IdleShop$

$$
\begin{array}{l}
x! : nil \mid talk \\
\hline
x! = transferableOperation \\
transferableOperation' = nil
\end{array}
$$

$in\_talk$

$$
\begin{array}{l}
\Delta(vmId, message) \\
y?, z? : \mathbb{Z} \\
\hline
y? = message' \\
z? = vmId'
\end{array}
$$

Figure 4.9: $\pi$-OZ specification of the *ActiveShop* using non-atomic reaction.

$$
\begin{array}{|l}
\hline
\textit{IdleShop}(id : \mathbb{Z}) \\
\\
\textit{chan ex\_switch} : nil \mid talk \\
\textit{chan local in\_switch} : nil \mid talk \\
\\
\textit{IdleShop\_PI} = \\
\qquad \textit{ex\_switch}(x, \textit{ex\_talk}).\textit{in\_switch} < x > .\textit{ActiveShop\_PI} \\
\\
\quad \begin{array}{|l}
\hline
\textit{self}, \textit{vmId}, \textit{message} : \mathbb{Z} \\
\textit{transferableOperation} : nil \mid talk \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\hline
\textsc{Init} \\
\textit{self} = id \\
\textit{transferableOperation} = nil \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\hline
\textit{in\_switch}\_\_\ \textit{then ActiveShop} \\
\Delta(\textit{transferableOperation}) \\
x? : nil \mid talk \\
\hline
x? = \textit{transferableOperation}' \\
\hline
\end{array} \\
\hline
\end{array}
$$

Figure 4.10: $\pi$-OZ specification of the *IdleShop* using non-atomic reaction.

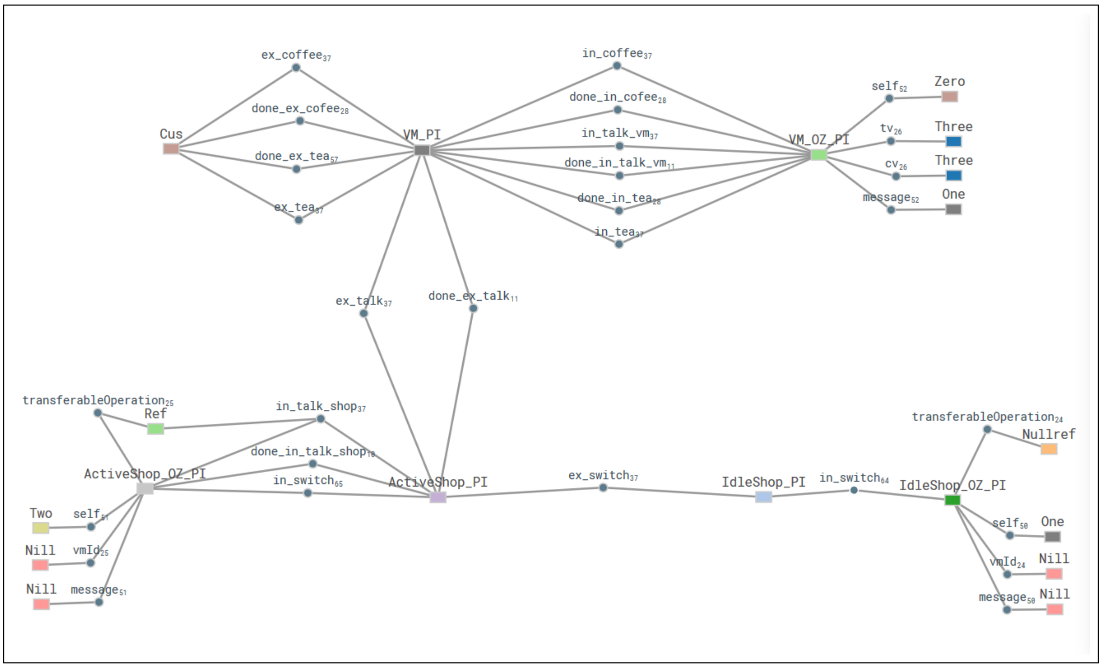



Figure 4.11: system consisting of a customer, vending machine and two shops

Additionally, Listing 4.1 shows the direct implementation of the $\pi$-part of *VM* specification shown in Figure 4.8 using ABC code. We can notice that: ex_coffee.'in_coffee.done_in_cofee.'done_ex_cofee.VM_PI... exactly reflects the $\pi$-part of Figure 4.8, where the parameters are removed for clarity.

```
agent VM_PI(ex_coffee,in_coffee,ex_tea,in_tea,done_ex_cofee,
   done_ex_tea,done_in_cofee,done_in_tea,ex_talk,in_talk,
   done_ex_talk,done_in_talk) =

 ex_coffee.'in_coffee.done_in_cofee.'done_ex_cofee.VM_PI(
    ex_coffee,in_coffee,ex_tea,in_tea,done_ex_cofee,
    done_ex_tea,done_in_cofee,done_in_tea,ex_talk,in_talk,
    done_ex_talk,done_in_talk)

+  ex_tea.'in_tea.done_in_tea.'done_ex_tea.VM_PI(ex_coffee,
   in_coffee,ex_tea,in_tea,done_ex_cofee,done_ex_tea,
   done_in_cofee,done_in_tea,ex_talk,in_talk,done_ex_talk,
   done_in_talk)

+  in_talk(r_c,m_c).'ex_talk<r_c,m_c>.done_ex_talk.'
   done_in_talk.VM_PI(ex_coffee,in_coffee,ex_tea,in_tea,
   done_ex_cofee,done_ex_tea,done_in_cofee,done_in_tea,
   ex_talk,in_talk,done_ex_talk,done_in_talk)
```

Listing 4.1: VM ($\pi$-part) in ABC code.

Listing 4.2 shows the direct implementation of the OZ-part of *VM* specification shown in Figure 4.8 using ABC code.

```
agent VM_OZ_PI_Init(in_coffee,in_tea,done_in_cofee,
   done_in_tea,in_talk,done_in_talk) =
 (^self,cv,tv,message) (VM_OZ_PI(self,in_coffee,in_tea,
    done_in_cofee,done_in_tea,in_talk,done_in_talk,cv,tv,
    message) | Zero(self) | Three(cv) | Three(tv) | One(
    message))



agent VM_OZ_PI(self,in_coffee,in_tea,done_in_cofee,
```

```
    done_in_tea,in_talk,done_in_talk,cv,tv,message) =

 in_coffee.(^res_t,res_f) (VM_Condition_coffee(self,
     in_coffee,in_tea,in_talk,cv,tv,message,res_t,res_f) |
     VM_Condition_IF_Else_coffee(self,in_coffee,in_tea,
     done_in_cofee,done_in_tea,in_talk,done_in_talk,cv,tv,
     message,res_t,res_f))

 + in_tea.(^res_t,res_f) (VM_Condition_tea(self,in_coffee,
     in_tea,in_talk,cv,tv,message,res_t,res_f) |
     VM_Condition_IF_Else_tea(self,in_coffee,in_tea,
     done_in_cofee,done_in_tea,in_talk,done_in_talk,cv,tv,
     message,res_t,res_f))

 + (^m_c,m_done,r_c,r_done) ( (m_done.r_done.'in_talk<r_c,
     m_c>.done_in_talk.VM_OZ_PI(self,in_coffee,in_tea,
     done_in_cofee,done_in_tea,in_talk,done_in_talk,cv,tv,
     message)) | Copy(message,m_c,m_done) | Copy(self,r_c,
     r_done))
```

Listing 4.2: VM (OZ-part) in ABC code.

Listing 4.3 shows the direct implementation of the π-part | OZ-part of *VM* specification shown in Figure 4.8 using ABC code.

```
agent VM(ex_coffee,ex_tea,done_ex_cofee,done_ex_tea,ex_talk,
    done_ex_talk) =
(^in_coffee,in_tea,done_in_cofee,done_in_tea,in_talk,
    done_in_talk)
(
 VM_OZ_PI_Init(in_coffee,in_tea,done_in_cofee,done_in_tea,
     in_talk,done_in_talk)
|
 VM_PI(ex_coffee,in_coffee,ex_tea,in_tea,done_ex_cofee,
     done_ex_tea,done_in_cofee,done_in_tea,ex_talk,in_talk,
     done_ex_talk,done_in_talk)
)
```

Listing 4.3: the combination $\pi$-OZ of VM in ABC code.

Listing 4.4 shows a part of the direct implementation of the system consisting of a customer, vending machine, and two shops shown in Figure 4.11 using ABC code. For the full code please see the appendix.

```
agent System = (^ex_coffee,done_ex_cofee,in_coffee,
    done_in_cofee,ex_tea,done_ex_tea,in_tea,done_ex_tea,
    done_in_tea,ex_switch,in_switch,ex_talk,in_talk_vm,
    in_talk_shop,done_ex_talk,done_in_talk_vm,
    done_in_talk_shop)

(
VM(ex_coffee,ex_tea,done_ex_cofee,done_ex_tea,ex_talk,
    done_ex_talk)
|
Cus(ex_coffee,ex_tea,done_ex_cofee,done_ex_tea)
|
IdleShop(ex_switch)
|
ActiveShop(ex_switch,ex_talk,done_ex_talk)
)
```

Listing 4.4: the system consisting of: customer, vending machine and two shops in ABC code.

# 5 Refinement

To study the refinement of $\pi$-calculus processes we will use the big-step trace semantics defined in [Gi14], where the set of all traces is defined as follow:

$$\mathtt{Traces} =_{\text{def}} \mathtt{seq}(\mathtt{Act} \setminus \{\tau\})$$

To abstract from the replacement of bound names, we use the equivalence class, denoted by $[P]$, which refers to all the processes obtained from $P$ by $\alpha$-conversion. Intuitively, an equivalence class represents all the processes that have the same behavior pattern.

An example of $\alpha$-conversion is: let $P_1 =_{\text{def}} \underline{\mathtt{new}}\, a\ \overline{a}\langle c\rangle$ and $P_2 =_{\text{def}} \underline{\mathtt{new}}\, b\ \overline{b}\langle c\rangle$, then $P_1 =_\alpha P_2$ with $\alpha = \{{}^{y}/_{a}\}$ where $y = b$.

The set of all equivalence classes is denoted by $\mathcal{P}^\pi_\alpha$

To determine the traces of a processes $P$ we use:

$$\mathcal{T}([P]) =_{\text{def}} \{t \in \mathtt{Traces} \mid \exists\, [Q] \in \mathcal{P}^\pi_\alpha : [P] \overset{t}{\Rightarrow} [Q]\} \tag{5.1}$$

The big-step semantics uses an early instantiation principle[1], and its results seem to be valid to our study.

The main result in [Gi14] is the following property:

$$([P]\,, [Q]) \in \mathcal{S} \Rightarrow [Q]\ \sqsubseteq_{\mathcal{T}}\ [P] \tag{5.2}$$

Property 5.2 reads: $[Q]$ strongly simulates[2] $[P]$ implies $[P]$ refines $[Q]$ in trace model,i.e. $[P]$ has less behavior than $[Q]$, where:

$$[Q]\ \sqsubseteq_{\mathcal{T}}\ [P] \Leftrightarrow \mathcal{T}([P]) \subseteq \mathcal{T}([Q])$$

[1]The early instantiation principl means that the bound name in an input prefix is instantiated directly when the input transition is inferred.

[2]Strong simulation considers $\tau$ actions as defined in Section 2.1.6

However, the work in [Gi14] was limited to recursion-free processes: "The limitation to recursion-free processes depends on the circumstances that we neither have any fix-point algorithm up to now nor showed that one existse"[3]. In this work we assume the existence of a fix-point algorithm and that Property 5.2 also applies to recursive processes. Seeking simplicity and preciseness we decided to avoid the concept of equivalence classes and to introduce the very strong simulation.

## 5.1 Very strong simulation

We noticed that the ABC tool changes the bound names during checking the simulation, which is not the case in the definition of strong simulation Definition 5.1.1. Thus, for clarity, we introduce the definition of the very strong simulation, which considers the use of the same bound names during the simulation checking.

**Definition 5.1.1 (Very strong simulation)** A relation $\mathcal{S}_v \subseteq \mathcal{P}^\pi \times \mathcal{P}^\pi$ is called a *very strong simulation*, if $(P, Q) \in \mathcal{S}_v$ implies that

$$P \xrightarrow{\alpha} P' \Rightarrow \exists\, Q' \in \mathcal{P}^\pi : Q \xrightarrow{\beta} Q' \wedge \alpha = \beta \wedge (P', Q') \in \mathcal{S}_v. \qquad \triangle$$

where $\alpha = \beta$ means $fn(\beta) = fn(\alpha) \wedge bn(\beta) = bn(\alpha)$

Additionally, we assume that Property 5.2 holds also for the very strong simulation without considering the equivalence classes. Formally:

$$(P, Q) \in \mathcal{S}_v \Rightarrow Q \sqsubseteq_{\mathcal{T}} P \tag{5.3}$$

Since the trace refinement does not say too much about the behavior of processes, we propose to use the Failure-Refinement model, originally introduced for CSP. In the next section we will define the failure-refinement for $\pi$-calculus processes and show that the very strong simulation does not imply failure-refinement. Thus, later in Section 5.3, we will introduce the Acceptance-Refinement model for $\pi$-calculus processes and show that the very strong simulation implies acceptance-refinement.

[3] [Gi14], page 8.

## 5.2 Failure-Refinement

We check whether the very strong simulation implies failure-refinement. We found that this is not the case. We start by defining the failure of a process. The pair $(t, X)$ is called a *failure*, where $t$ is a trace and $X$ is a set of impossible next actions. Any process $P$ is assigned a set of failures $F$. Formally, this means:

$$\mathcal{F}(P) =_{\text{def}} \{(t, X) \mid \exists\, Q \in \mathcal{P}^\pi : P \stackrel{t}{\Rightarrow} Q \wedge Q\mathit{ref}X \wedge X \in \mathit{Refusals}\} \tag{5.4}$$

where: $\mathit{Refusals} =_{\text{def}} \mathbb{P}(\texttt{Act} \setminus \{\tau\})$.

We can define the failure-refinement of $\pi$-calculus processes as follow:

**Definition 5.2.1 (Failure refinement)** Let $P, Q \in \mathcal{P}^\pi$, then $P$ is a *failure refinement* of $Q$ iff the inverse set inclusion of traces and failure holds:

$$Q \sqsubseteq_{\mathcal{F}} P \Leftrightarrow \mathcal{T}(P) \subseteq \mathcal{T}(Q) \wedge \mathcal{F}(P) \subseteq \mathcal{F}(Q) \tag{5.5}$$

△

From Property 5.3 and Definition 5.3.1 we can drive the following Corollary:

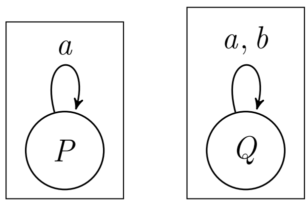
**Corollary 5.2.1 (Simulation and Failure refinement)** *Let $P, Q \in \mathcal{P}^\pi$ processes. If $Q$ very strongly simulates $P$, then $P$ refines $Q$ in Failure-Refinement model. Formally written:*

$$(P, Q) \in \mathcal{S}_v \not\Rightarrow Q \sqsubseteq_{\mathcal{F}} P \tag{5.6}$$

□

Due to performance issue, we will limit the check to behavior part. That is, we are not comparing the combination $\pi$-OZ, but only the $\pi$ part without data, as shown in Figure 5.1.

**Proof:** by counter example. Assume that $(P, Q) \in \mathcal{S}_v \not\Rightarrow Q \sqsubseteq_{\mathcal{F}} P$ and let $P =_{\text{def}} \overline{a}\langle\rangle.P$ and $Q =_{\text{def}} \overline{a}\langle\rangle.Q + \overline{b}\langle\rangle.Q$ shown in Figure 5.1.




(a) $P$ (b) $Q$

Figure 5.1: $P$ and $Q$

It is clear that $Q$ very strongly simulates $P$. This result implies, according to Corollary 5.2.1, that $P$ refines $Q$ in the failure model, thus we need to show that $\mathcal{T}(P) \subseteq \mathcal{T}(Q) \wedge \mathcal{F}(P) \subseteq \mathcal{F}(Q)$.

- For $\mathcal{T}(P) \subseteq \mathcal{T}(Q)$: we need to determine the traces of $Q$ and $P$ shown in Figure 5.1. According to 5.1:

$$\mathcal{T}(Q) =_{\text{def}} \{a(), b()\}^*$$

$$\mathcal{T}(P) =_{\text{def}} \{a()\}^*$$

  It is clear that $\mathcal{T}(P) \subseteq \mathcal{T}(Q)$ holds.

- For $\mathcal{F}(P) \subseteq \mathcal{F}(Q)$: let $\epsilon$ be the empty trace, then

$$\mathcal{F}(Q) =_{\text{def}} \{(\epsilon, \{\}), \ldots\}^*$$

$$\mathcal{F}(P) =_{\text{def}} \{(\epsilon, \{b()\}), \ldots\}^*$$

  It is clear that $\mathcal{F}(P) \nsubseteq \mathcal{F}(Q)$, thus $Q \not\sqsubseteq_{\mathcal{F}} P$.

  So, very strong simulation does not imply failure-refinement. Thus, in the next section we will introduce the Acceptance-Refinement model and use it instead of the Failure-Refinement model. ■

## 5.3 Acceptance-Refinement

To compare $\pi$-calculus processes we need to define the Acceptance-Refinement and relate it to the very strong simulation. We start by defining the *acceptance* pair of a process. The pair $(t, Y)$ is called a acceptance pair, where $t$ is a trace and $Y$ is a set of all possible next actions. Any process $P$ is assigned a set of acceptance pairs $AC$. Formally, this means:

$$\mathcal{AC}(P) =_{\text{def}} \{(t, Y) \mid \exists\, Q \in \mathcal{P}^{\pi} : P \stackrel{t}{\Rightarrow} Q \wedge Y \in AA_{\alpha}\} \tag{5.7}$$

where: $AA =_{\text{def}} \mathbb{P}(\texttt{Act} \setminus \{\tau\})$.

We can define the Acceptance-refinement of $\pi$-calculus processes as follow:

**Definition 5.3.1 (Acceptance refinement)** Let $P, Q \in \mathcal{P}^{\pi}$, then $P$ is a *acceptance refinement* of $Q$ iff the inverse set inclusion of traces and acceptances holds:

$$Q \sqsubseteq_{\mathcal{AC}} P \Leftrightarrow \mathcal{T}(P) \subseteq \mathcal{T}(Q) \wedge \mathcal{AC}(P) \subseteq \mathcal{AC}(Q) \tag{5.8}$$

$\triangle$

From Property 5.2 and Definition 5.3.1 we can drive the following Corollary:

**Corollary 5.3.1 (Simulation and Acceptance refinement)** *Let $P, Q \in \mathcal{P}^{\pi}$ processes. If $Q$ very strongly simulates $P$, then $P$ refines $Q$ in Acceptance-Refinement model. Formally written:*

$$(P, Q) \in \mathcal{S}_v \Rightarrow Q \sqsubseteq_{\mathcal{AC}} P \tag{5.9}$$

*holds.* $\square$

**Proof:** Let $(P, Q) \in \mathcal{S}_v$, then $\mathcal{T}(P) \subseteq \mathcal{T}(Q) \wedge \mathcal{AC}(P) \subseteq \mathcal{AC}(Q)$ holds, Since:

- For $\mathcal{T}(P) \subseteq \mathcal{T}(Q)$: it holds using Property 5.3.
- For $\mathcal{AC}(P) \subseteq \mathcal{AC}(Q)$: we need to show that,
  $\forall (t_P, Y_P) \in \mathcal{AC}(P)$ *then* $\exists (t_Q, Y_Q) \in \mathcal{AC}(Q) : t_p = t_Q \wedge Y_P \subseteq Y_Q$
  - $t_p = t_Q$ holds using Property 5.2.
  - $Y_P \subseteq Y_Q$ holds, since $Q$ very strongly simulates $P$ means that $Q$ can do all the actions that $P$ can do after any trace $t$. $\blacksquare$

### Acceptance-Refinement use case:

We will show that if $Q$ shown in Figure 5.1 very strongly simulates $P$, then $P$ refines $Q$ in acceptance-refinement model.

Listing **??** showed that $Q$ strongly simulates $P$. This result implies, according to Corollary 5.3.1, that $P$ refines $Q$ in the acceptance-refinement model. Thus, we need to show that $\mathcal{T}(P) \subseteq \mathcal{T}(Q) \wedge \mathcal{AC}(P) \subseteq \mathcal{AC}(Q)$.

- For $\mathcal{T}(P) \subseteq \mathcal{T}(Q)$: previously we showed that it holds.
- For $\mathcal{AC}(P) \subseteq \mathcal{AC}(Q)$: let $\epsilon$ be the empty trace, then

$$\mathcal{AC}(Q) =_{\text{def}} \{(\epsilon, \{tea(), coffee(), talk <>\}), \ldots\}^*$$

$$\mathcal{A}C(P) =_{\text{def}} \{(\epsilon, \{\mathit{coffee}(), \mathit{talk} <>\}), \ldots\}^*$$

It is clear that $\mathcal{A}C(P) \subseteq \mathcal{A}C(Q)$ holds, thus $Q \sqsubseteq_{\mathcal{A}C} P$ holds.

## 5.4 New:

| $\alpha$ | denotation | $\mathtt{n}(\alpha)$ | $\mathtt{bn}(\alpha)$ | $\mathtt{fn}(\alpha)$ | $\sigma(\alpha)$ | $\overline{\alpha}$ |
|---|---|---|---|---|---|---|
| $\tau$ | internal | $\varnothing$ | $\varnothing$ | $\varnothing$ | $\tau$ | $\tau$ |
| $a\,x$ | input | $\{a, x\}$ | $\varnothing$ | $\{a, x\}$ | $\sigma(a)\,\sigma(x)$ | $\overline{a}\langle x\rangle$ |
| $\overline{a}\langle x\rangle$ | output | $\{a, x\}$ | $\varnothing$ | $\{a, x\}$ | $\overline{\sigma(a)}\langle\sigma(x)\rangle$ | $a\,x$ |
| $\overline{a}(x)$ | bound output | $\{a, x\}$ | $\{x\}$ | $\{a\}$ | $\overline{\sigma(a)}(x)$ | $a\,x$ |

Figure 5.2: Free and bound names of actions.

$$\underline{E\text{-}TAU} : \frac{}{[\tau.P] \xrightarrow{\tau} [P]} \qquad \underline{E\text{-}CALL} : \frac{}{[A\langle\vec{v}\rangle] \xrightarrow{\tau} [P \subseteq \vec{v}\vec{w}]} \quad A(\vec{w}) =_{\text{def}} P$$

$$\underline{E\text{-}OUT} : \frac{}{[\overline{x}\langle y\rangle.P] \xrightarrow{\overline{x}\langle y\rangle} [P]} \qquad \underline{E\text{-}IN} : \frac{}{[x(z).P] \xrightarrow{x\,y} [P \subseteq yz]}$$

$$\underline{E\text{-}SUM_L} : \frac{[P] \xrightarrow{\alpha} [P']}{[P + Q] \xrightarrow{\alpha} [P']} \qquad \underline{E\text{-}RES} : \frac{[P] \xrightarrow{\alpha} [P']}{[\underline{\mathtt{new}}\, z\ P] \xrightarrow{\alpha} [\underline{\mathtt{new}}\, z\ P']} \quad z \notin n(\alpha)$$

$$\underline{E\text{-}PAR_L} : \frac{[P] \xrightarrow{\alpha} [P']}{[P \mid Q] \xrightarrow{\alpha} [P' \mid Q]} \quad \mathtt{bn}(\alpha) \cap \mathtt{fn}(Q) = \varnothing$$

$$\underline{E\text{-}OPEN} : \frac{[P] \xrightarrow{\overline{x}\langle z\rangle} [P']}{[\underline{\mathtt{new}}\, z\ P] \xrightarrow{\overline{x}(z)} [P']} \quad z \neq x \qquad \underline{E\text{-}COM_L} : \frac{[P] \xrightarrow{\overline{x}\langle y\rangle} [P'] \quad [Q] \xrightarrow{x\,y} [Q']}{[P \mid Q] \xrightarrow{\tau} [P' \mid Q']}$$

$$\underline{E\text{-}CLOSE_L} : \frac{[P] \xrightarrow{\overline{x}(z)} [P'] \quad [Q] \xrightarrow{x\,z} [Q']}{[P \mid Q] \xrightarrow{\tau} [\underline{\mathtt{new}}\, z\,(P' \mid Q')]} \quad z \notin \mathtt{fn}(Q)$$

Figure 5.3: The *early transition system* [SW01].

$$bn(P) =_{\text{def}} \{n \mid (n is\ restricted) \vee (\exists\, \alpha \in Act : n = obj(\alpha))\}$$
$$fn(P) =_{\text{def}} \{n \mid (n appears\ in P) \wedge (n \notin bn(P))\}$$

$Acc =_{\text{def}} \{\alpha \mid (\alpha = \tau) \vee (sub(\alpha) \in fn(P) \wedge obj(\alpha) = signal) \vee (sub(\alpha) \in fn(P) \wedge obj(\alpha) \in fn(P)) \vee (sub(\alpha) \in fn(P) \wedge obj(\alpha) \in bn(P))\}$

let $P =_{\text{def}}$ <u>`new`</u> $c\ (\overline{a}\langle c\rangle.0 + \overline{a}\langle c\rangle.P) + \tau.0 + \tau.P$, then the traces of $P$ and $[P]$ are:

let $Q =_{\text{def}}$ <u>`new`</u> $c, k\ (\overline{a}\langle c\rangle + \overline{a}\langle k\rangle)$, then the traces of $Q$ and $[Q]$ are:

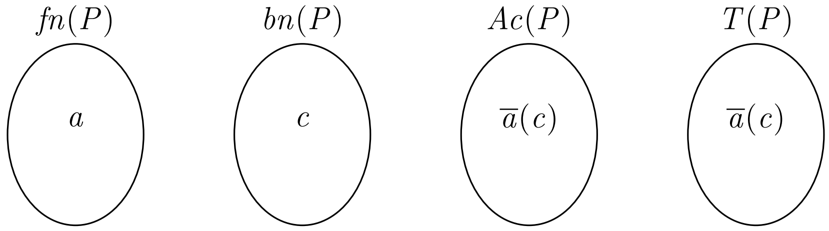



Figure 5.4: traces

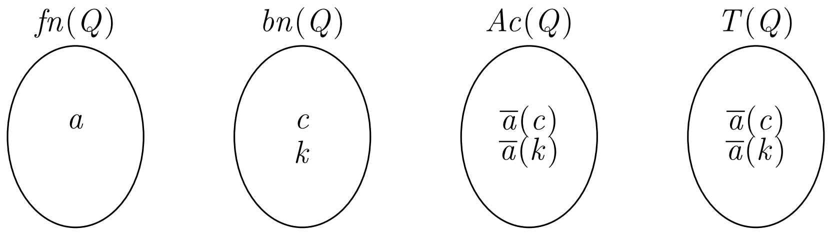



Figure 5.5: traces

# 6 Conclusion and future work

In this thesis we investigated the ransformational semantics of the combination $\pi$-OZ for mobile processes with data.

The aim of this work is to combine the OZ specification with $\pi$-calculus spesification, and to tranform the combination into a $\pi$-calculus processes, in a similar way to CSP-OZ [Ol18] approach. Unfortunately, we found out that the transformation is Cumbersome, since $\pi$-calculus has only elementary constructs and not suitable to express complex constructs like OZ class constructs. On the one hand, we showed how to integrate a $\pi$-calculus process, describing the desired sequence of behavior, into an OZ class to form the $\pi$-OZ combination. On the other hand, we explained how to transform the combination $\pi$-OZ into $\pi$-calculus process through transforming OZ class constructs value, state variable, state schema, initial state schema and operation schema into $\pi$-calculus processes and names accompanied by the processes of the desired sequence of behavior using the parallel operator. In spite of that, we introduced the Failure-Refinement model for $\pi$-calculus and showed that the strong simulation does not imply the failure-refinement. Finally, we introduced the Acceptance-Refinement model and showed that the strong simulation implies acceptance-refinement.

The elementary nature of the $\pi$-calculus is a a good start for future work through introducing a state-full $\pi$-calculus, which is an extension of $\pi$-calculus to supports data, data types, variable and transition conditions. This will ease the mapping between OZ and $\pi$-calculus constructs. Additionaly, a tool can be developed to visualize the state-full $\pi$-calculus in a similar way to Stargazer[Star]. Furthermore, it would be good to extend the ABC simulation-checker [ABC] to support simulation checking and acceptance-refinement verification of the state-full $\pi$-calculus. Finally, it would be interesting to extend [Gi14] trace semantics to support recursive processes through developing a fixed-point algorithm for the recursive processes. This will facilitate the study of the composition properties of the Acceptance-Refinement model, which will permit to study the possibility of introducing an Acceptance-Divergence model and to explore its composition properties.

# 7 Appendix

## 7.1 Addition

```
agent Zero(a) = a(tt,ff).'ff.'ff.Zero(a) + a.0
agent One(a) = a(tt,ff).'tt.'ff.One(a) + a.0
agent Two(a) = a(tt,ff).'ff.'tt.Two(a) + a.0
agent Three(a) = a(tt,ff).'tt.'tt.Three(a) + a.0
```

Listing 7.1: 0,1,2,3 as $\pi$-calculus processes.

```
agent FullAdderWait(t1,f1,t2,f2,cin_t,cin_f,cout3_t,cout3_f,
    s3_t,s3_f) = \
cin_t.('cin_t | FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout3_t,
    cout3_f,s3_t,s3_f)) \
+ cin_f.('cin_f | FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout3_t,
    cout3_f,s3_t,s3_f))

agent FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout_t,cout_f,s2_t,
    s2_f) = \
(^t1a,f1a,t1b,f1b,t2a,f2a,t2b,f2b,c1_t,c1_f,s1_t,s1_f,c2_t,
    c2_f,s1_ta,s1_fa,s1_tb,s1_fb,cin_ta,cin_fa,cin_tb,cin_fb)
     \
( \
HalfAdder(t1,f1,t1a,f1a,t1b,f1b,t2,f2,t2a,f2a,t2b,f2b,c1_t,
    c1_f,s1_t,s1_f) \
| HalfAdder(s1_t,s1_f,s1_ta,s1_fa,s1_tb,s1_fb,cin_t,cin_f,
    cin_ta,cin_fa,cin_tb,cin_fb,c2_t,c2_f,s2_t,s2_f) \
| OR(c1_t,c1_f,c2_t,c2_f,cout_t,cout_f) \
)
```

```
agent HalfAdder(t1,f1,t1a,f1a,t1b,f1b,t2,f2,t2a,f2a,t2b,f2b,
    c_t,c_f,s_t,s_f) = (Repeate(t1,f1,t1a,f1a,t1b,f1b) |
    Repeate(t2,f2,t2a,f2a,t2b,f2b) | AND(t1a,f1a,t2a,f2a,c_t,
    c_f) | XOR(t1b,f1b,t2b,f2b,s_t,s_f))

agent AND(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_f + t2.'o_f) + t1
    .(f2.'o_f + t2.'o_t)
agent NAND(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_t + t2.'o_t) +
    t1.(f2.'o_t + t2.'o_f)
agent OR(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_f + t2.'o_t) + t1
    .(f2.'o_t + t2.'o_t)
agent NOR(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_t + t2.'o_f) + t1
    .(f2.'o_f + t2.'o_f)
agent XOR(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_f + t2.'o_t) + t1
    .(f2.'o_t + t2.'o_f)
agent XNOR(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_t + t2.'o_f) +
    t1.(f2.'o_f + t2.'o_t)

agent Send(a) = 'a
agent Neg(tt,ff) = tt.'ff + ff.'tt
agent Repeate(tt,ff,ta,fa,tb,fb) = tt.('ta | 'tb) + ff.('fa
    | 'fb)
```

Listing 7.2: Gates.

```
agent Example = (^a,b,c) (Two(a) | One(b) | Add(a,b,c))

agent Add(a,b,c)= (^t1,f1,t2,f2) ('a<t1,f1>.'b<t2,f2>.(^
    cin_t,cin_f,cout_t,cout_f,s2_t,s2_f,cout3_t,cout3_f,s3_t,
    s3_f) ( \
FullAdderWait(t1,f1,t2,f2,cin_t,cin_f,cout_t,cout_f,s2_t,
    s2_f) \
| 'cin_f \
| FullAdderWait(t1,f1,t2,f2,cout_t,cout_f,cout3_t,cout3_f,
    s3_t,s3_f) \
| Result(s2_t,s2_f,s3_t,s3_f,cout3_t,cout3_f,c) \
```

```
))

agent Result(s0_t,s0_f,s1_t,s1_f,c_t,c_f,res) = \
s0_f.(s1_f.(c_f.Zero(res) + c_t.OverFlow) + s1_t.(c_f.Two(
    res)+ c_t.OverFlow)) \
+ s0_t.(s1_f.(c_f.One(res) + c_t.OverFlow) + s1_t.(c_f.Three
    (res)+ c_t.OverFlow))

agent OverFlow = 0
```

Listing 7.3: Adder.

```
agent Example = (^a,b,c) (Three(a) | One(b) | Sub(a,b,c))

(* the trick is to invert t2,f2 places to mimic the Inverter
     *)
agent Sub(a,b,c) = (^t1,f1,t2,f2) ('a<t1,f1>.'b<t2,f2>.(^
    cin_t,cin_f,cout_t,cout_f,s2_t,s2_f,cout3_t,cout3_f,s3_t,
    s3_f) ( \
FullAdderWait(t1,f1,f2,t2,cin_t,cin_f,cout_t,cout_f,s2_t,
    s2_f) \
| 'cin_t \
| FullAdderWait(t1,f1,f2,t2,cout_t,cout_f,cout3_t,cout3_f,
    s3_t,s3_f) \
| Result_S(s2_t,s2_f,s3_t,s3_f,cout3_t,cout3_f,c) \
))

agent Result_S(s0_t,s0_f,s1_t,s1_f,c_t,c_f,res) = \
s0_f.(s1_f.(c_t.Zero(res) + c_f.ErrNegResult ) + s1_t.(c_t.
    Two(res) + c_f.ErrNegResult )) \
+ s0_t.(s1_f.(c_t.One(res) + c_f.ErrNegResult ) + s1_t.(c_t.
    Three(res) + c_f.ErrNegResult))



agent ErrNegResult = 0
```

```
agent FullAdderWait(t1,f1,t2,f2,cin_t,cin_f,cout3_t,cout3_f,
    s3_t,s3_f) = \
cin_t.('cin_t | FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout3_t,
    cout3_f,s3_t,s3_f)) \
+ cin_f.('cin_f | FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout3_t,
    cout3_f,s3_t,s3_f))



agent FullAdder(t1,f1,t2,f2,cin_t,cin_f,cout_t,cout_f,s2_t,
    s2_f) = \
(^t1a,f1a,t1b,f1b,t2a,f2a,t2b,f2b,c1_t,c1_f,s1_t,s1_f,c2_t,
    c2_f,s1_ta,s1_fa,s1_tb,s1_fb,cin_ta,cin_fa,cin_tb,cin_fb)
     \
( \
HalfAdder(t1,f1,t1a,f1a,t1b,f1b,t2,f2,t2a,f2a,t2b,f2b,c1_t,
    c1_f,s1_t,s1_f) \
| HalfAdder(s1_t,s1_f,s1_ta,s1_fa,s1_tb,s1_fb,cin_t,cin_f,
    cin_ta,cin_fa,cin_tb,cin_fb,c2_t,c2_f,s2_t,s2_f) \
| OR(c1_t,c1_f,c2_t,c2_f,cout_t,cout_f) \
)

agent HalfAdder(t1,f1,t1a,f1a,t1b,f1b,t2,f2,t2a,f2a,t2b,f2b,
    c_t,c_f,s_t,s_f) = (Repeate(t1,f1,t1a,f1a,t1b,f1b) |
    Repeate(t2,f2,t2a,f2a,t2b,f2b) | AND(t1a,f1a,t2a,f2a,c_t,
    c_f) | XOR(t1b,f1b,t2b,f2b,s_t,s_f))
```

Listing 7.4: Subtractor.

```


agent Example = (^a,b,g,e,l) (Three(a) | Two(b) | Compare(a,
    b,g,e,l) | If_Else(g,e,l))

agent If_Else(g,e,l) = g.Greater + e.Equal + l.Less

agent Greater = 0
agent Equal = 0
```

```
agent Less = 0

agent Compare(a,b,g,e,l) = (^ta,fa,tb,fb,l_t,l_f,e_t,e_f,g_t
    ,g_f) ('a<ta,fa>.'b<tb,fb>.(^tb1,fb1,tb2,fb2,o_xor_t,
    o_xor_f,o_xor_1t,o_xor_1f,o_xor_2t,o_xor_2f,o_nand_1_t,
    o_nand_1_f,o_nand_1_1t,o_nand_1_1f,o_nand_1_2t,
    o_nand_1_2f)( \
Repeate(tb,fb,tb1,fb1,tb2,fb2) \
| XOR(ta,fa,tb1,fb1,o_xor_t,o_xor_f) \
| Repeate(o_xor_t,o_xor_f,o_xor_1t,o_xor_1f,o_xor_2t,
    o_xor_2f) \
| NAND(tb2,fb2,o_xor_2t,o_xor_2f,o_nand_1_t,o_nand_1_f) \
| Repeate(o_nand_1_t,o_nand_1_f,o_nand_1_1t,o_nand_1_1f,
    o_nand_1_2t,o_nand_1_2f) \
| Compare_3(a,b,l_t,l_f,e_t,e_f,g_t,g_f,ta,fa,tb,fb,
    o_nand_1_1t,o_nand_1_1f,o_nand_1_2t,o_nand_1_2f,o_xor_1t,
    o_xor_1f,g,e,l) \
))


agent Compare_3(a,b,l_t,l_f,e_t,e_f,g_t,g_f,ta,fa,tb,fb,
    o_nand_1_1t,o_nand_1_1f,o_nand_1_2t,o_nand_1_2f,o_xor_1t,
    o_xor_1f,g,e,l) = \
o_nand_1_1t.Compare_4(a,b,l_t,l_f,e_t,e_f,g_t,g_f,ta,fa,tb,
    fb,o_nand_1_2t,o_nand_1_2f,o_xor_1t,o_xor_1f,g,e,l) \
+ o_nand_1_1f.Compare_4(a,b,l_t,l_f,e_t,e_f,g_t,g_f,ta,fa,tb
    ,fb,o_nand_1_2t,o_nand_1_2f,o_xor_1t,o_xor_1f,g,e,l)


agent Compare_4(a,b,l_t,l_f,e_t,e_f,g_t,g_f,ta,fa,tb,fb,
    o_nand_1_2t,o_nand_1_2f,o_xor_1t,o_xor_1f,g,e,l) = (^tb1,
    fb1,tb2,fb2,o_xnor_t,o_xnor_f,o_xnor_1t,o_xnor_1f,
    o_xnor_2t,o_xnor_2f,o_nor_1_t,o_nor_1_f,o_nand_2_t,
    o_nand_2_f,o_nand_2_1t,o_nand_2_1f,o_nand_2_2t,
    o_nand_2_2f,o_nor_2_t,o_nor_2_f,o_xor_2_t,o_xor_2_f,
```

```
    o_nor_2_1t,o_nor_2_1f,o_nor_2_2t,o_nor_2_2f,o_xor_2_1t,
    o_xor_2_1f,o_xor_2_2t,o_xor_2_2f,o_nor_3_t,o_nor_3_f,e2t,
    e2f,l2t,l2f)(\
Repeate(tb,fb,tb1,fb1,tb2,fb2) \
| XNOR(ta,fa,tb1,fb1,o_xnor_t,o_xnor_f) \
| Repeate(o_xnor_t,o_xnor_f,o_xnor_1t,o_xnor_1f,o_xnor_2t,
    o_xnor_2f) \
| NOR(tb2,fb2,o_xnor_1t,o_xnor_1f,o_nor_1_t,o_nor_1_f) \
| NAND(o_xnor_2t,o_xnor_2f,o_nand_1_2t,o_nand_1_2f,
    o_nand_2_t,o_nand_2_f) \
| Repeate(o_nand_2_t,o_nand_2_f,o_nand_2_1t,o_nand_2_1f,
    o_nand_2_2t,o_nand_2_2f) \
| NOR(o_nand_2_1t,o_nand_2_1f,o_xor_1t,o_xor_1f,o_nor_2_t,
    o_nor_2_f) \
| XOR(o_nor_1_t,o_nor_1_f,o_nand_2_2t,o_nand_2_2f,o_xor_2_t,
    o_xor_2_f) \
| Repeate(o_nor_2_t,o_nor_2_f,e_t,e_f,e2t,e2f) \
| Repeate(o_xor_2_t,o_xor_2_f,l_t,l_f,l2t,l2f) \
| NOR(e2t,e2f,l2t,l2f,g_t,g_f) \
| Compare_5(l_t,l_f,e_t,e_f,g_t,g_f,g,e,l) \
)
agent Compare_5(l_t,l_f,e_t,e_f,g_t,g_f,g,e,l) = g_t.
    Compare_6(g,e_f,l_f) + e_t.Compare_6(e,l_f,g_f)+ l_t.
    Compare_6(l,e_f,g_f)

agent Compare_6(a,b,c) = b.c.'a
```

Listing 7.5: Comparator.

```
agent Example = (^a1,a3) (List1(a1) | List2(a3) | Union(a1,
    a3))

agent List1(a1) = (^a0,b0,c0,d0,b1,c1) (Node(a1,b1,a0) | Ref
    (b1,c1) | Three(c1) | Node(a0,b0,d0) | Ref(b0,c0) | Zero(
    c0) | Nil(d0))
agent List2(a3) = (^a2,b2,c2,d1,b3,c3) (Node(a3,b3,a2) | Ref
```

```
    (b3,c3) | Three(c3) | Node(a2,b2,d1) | Ref(b2,c2) | Two(
    c2) | Nil(d1))

agent Union(a1,a3) = AddElement(a1,a3)
agent AddElement(a1,a3) = (^res_t,res_f,o) ( GetValue(a3,o)
    | CheckValue(a1,o,a3))
agent GetValue(r,o) = (^n,c) ('r<n,c>.(c(rv,l).'rv<n,c>.c(v)
    .'o<v,l> + n.'o))
agent CheckValue(k1,o,k2) =  o(v,l).(^res_t,res_f) (In(v,k1,
    res_t,res_f) | Append(v,k1,l,res_t,res_f)) + o
agent Append(v,k1,l,res_t,res_f) = res_t.AppendNo(k1,l) +
    res_f.AppendYes(v,k1,l)
agent AppendNo(a,l) = AddElement(a,l)
agent AppendYes(v,k1,l) = (^a,b,c) (Copy(v,c) | Node(a,b,k1)
     | Ref(b,c) | AddElement(a,l))
agent Copy(a,b) = (^tt,ff) ('a<tt,ff>.(ff.(ff.Zero(b) + tt.
    Two(b)) + tt.(tt.Three(b)+ ff.One(b))))
agent In(a,b,res_t,res_f) = (^n,c) (In_1(a,b,res_t,res_f,n,c
    ))
agent In_1(a,b,res_t,res_f,n,c) = 'b<n,c>.(c(r,l).'r<n,c>.
    In_4(a,b,res_t,res_f,n,c,r,l)  + n.'res_f)
agent In_4(a,b,res_t,res_f,n,c,r,l) = c(v).(^out_t,out_f) (
    IsEqual(a,v,out_t,out_f) | In_5(a,b,res_t,res_f,n,c,r,l,
    out_t,out_f))
agent In_5(a,b,res_t,res_f,n,c,r,l,out_t,out_f) = out_t.'
    res_t + out_f.In_1(a,l,res_t,res_f,n,c)
agent IsEqual(a,b,out_t,out_f) = (^t1,f1,t2,f2) ('a<t1,f1>.'
    b<t2,f2>.(^o_t,o_f) (IsEqual_4(a,b,out_t,out_f,t1,f1,t2,
    f2,o_t,o_f) | CompareBit( t1,f1,t2,f2,o_t,o_f)))
agent IsEqual_4(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) = o_t.(
    IsEqual_5(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) |
    CompareBit( t1,f1,t2,f2,o_t,o_f)) + o_f.IsEqualPassBit( a
    ,b,out_t,out_f,t1,f1,t2,f2)
agent IsEqualPassBit(a,b,out_t,out_f,t1,f1,t2,f2) = f1.(f2.'
    out_f + t2.'out_f) + t1.(f2.'out_f + t2.'out_f)
```

```
agent IsEqual_5(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) = o_t.'
    out_t + o_f.'out_f

agent CompareBit(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_t + t2.'
    o_f) + t1.(t2.'o_t + f2.'o_f)

agent Nullref(r) = r(n,c).('n.Nullref(r) + c(m).Ref(r,m) + n
    .Nullref(r))
agent Ref(r,v) = r(n,c).('c<v>.Ref(r,v) + c(m).Ref(r,m) + 'n
    .Nullref(r))
agent Nil(k) = k(n,c).'n.Nil(k)
agent Node(k,v,l) = k(n,c).'c<v,l>.Node(k,v,l)
```

Listing 7.6: Set union.

```
agent Example = (^a1,a5) (List1(a1) | List2(a5) | Subtract(
    a1,a5))

agent List1(a1) = (^a0,b0,c0,d0,b1,c1) (Node(a1,b1,a0) | Ref
    (b1,c1) | Three(c1) | Node(a0,b0,d0) | Ref(b0,c0) | Zero(
    c0) | Nil(d0))
agent List2(a5) = (^a2,b2,c2,d1,a3,b3,c3,a4,b4,c4,b5,c5) (
    Node(a5,b5,a4) | Ref(b5,c5) | Zero(c5) | Node(a4,b4,a3) |
     Ref(b4,c4) | One(c4) | Node(a3,b3,a2) | Ref(b3,c3) |
    Three(c3) | Node(a2,b2,d1) | Ref(b2,c2) | One(c2) | Nil(
    d1))

agent Subtract(a1,a3) = SubtElement(a1,a3)
agent SubtElement(a1,a3) = (^res_t,res_f,o) ( GetValue(a3,o)
     | CheckValue_S(a1,o,a3))

agent GetValue(r,o) = (^n,c) ('r<n,c>.(c(rv,l).'rv<n,c>.c(v)
    .'o<v,l> + n.'o))

agent CheckValue_S(k1,o,k2) =  o(v,l).(^res_t,res_f) (In(v,
    k1,res_t,res_f) | Check_In(v,k1,k2,l,res_t,res_f)) + o
```

```
agent Append(v,k1,l,res_t,res_f) = res_t.AppendNo(k1,l) +
   res_f.AppendYes(v,k1,l)
agent AppendNo(a,l) = AddElement(a,l)
agent AppendYes(v,k1,l) = (^a,b,c) (Copy(v,c) | Node(a,b,k1)
    | Ref(b,c) | AddElement(a,l))

agent Check_In(v,k1,k2,l,res_t,res_f) = res_t.RemoveYes(k1,
   k2,l) + res_f.RemoveNo(k1,l)
agent RemoveNo(k,l) = SubtElement(k,l)
agent RemoveYes(k1,k2,l) = 'k2.RemoveYes_1(k1,k2,l)
agent RemoveYes_1(k1,k2,l) = (^o) (GetValue(l,o) | FixIndex(
   o,k1,k2,l))
agent FixIndex(o,k1,k2,l) = o(v,l1).(^a,b,c,done) (Copy_S(v,
   c,done) | Node(k2,b,l1) | Ref(b,c) | done.'l.SubtElement(
   k1,l1))

agent Copy_S(a,b,done) = (^tt,ff) ('a<tt,ff>.(ff.(ff.(Zero(b
   ) | 'done) + tt.(Two(b) | 'done)) + tt.(tt.(Three(b) | '
   done)+ ff.(One(b) | 'done))))

agent In(a,b,res_t,res_f) = (^n,c) (In_1(a,b,res_t,res_f,n,c
   ))
agent In_1(a,b,res_t,res_f,n,c) = 'b<n,c>.(c(r,l).'r<n,c>.
   In_4(a,b,res_t,res_f,n,c,r,l)  + n.'res_f)

agent In_4(a,b,res_t,res_f,n,c,r,l) = c(v).(^out_t,out_f) (
   IsEqual(a,v,out_t,out_f) | In_5(a,b,res_t,res_f,n,c,r,l,
   out_t,out_f))
agent In_5(a,b,res_t,res_f,n,c,r,l,out_t,out_f) = out_t.'
   res_t + out_f.In_1(a,l,res_t,res_f,n,c)

agent IsEqual(a,b,out_t,out_f) = (^t1,f1,t2,f2) ('a<t1,f1>.'
   b<t2,f2>.(^o_t,o_f) (IsEqual_4(a,b,out_t,out_f,t1,f1,t2,
   f2,o_t,o_f) | CompareBit( t1,f1,t2,f2,o_t,o_f)))
```

```
agent IsEqual_4(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) = o_t.(
    IsEqual_5(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) |
    CompareBit( t1,f1,t2,f2,o_t,o_f)) + o_f.IsEqualPassBit( a
    ,b,out_t,out_f,t1,f1,t2,f2)

agent IsEqualPassBit(a,b,out_t,out_f,t1,f1,t2,f2) = f1.(f2.'
    out_f + t2.'out_f) + t1.(f2.'out_f + t2.'out_f)
agent IsEqual_5(a,b,out_t,out_f,t1,f1,t2,f2,o_t,o_f) = o_t.'
    out_t + o_f.'out_f

agent CompareBit(t1,f1,t2,f2,o_t,o_f) = f1.(f2.'o_t + t2.'
    o_f) + t1.(t2.'o_t + f2.'o_f)

agent Nullref(r) = r(n,c).('n.Nullref(r) + c(m).Ref(r,m) + n
    .Nullref(r))
agent Ref(r,v) = r(n,c).('c<v>.Ref(r,v) + c(m).Ref(r,m) + 'n
    .Nullref(r))
agent Nil(k) = k(n,c).'n.Nil(k)
agent Node(k,v,l) = k(n,c).'c<v,l>.Node(k,v,l)
```

Listing 7.7: Set subtraction.

# Bibliography


[Mil99] R. Milner. *Communicating and Mobile Systems: the $\pi$-Calculus.* Cambridge University Press, Cambridge, England, 1999.

[SW01] D. Sangiorgi and D. Walker. *The $\pi$-Calculus: a Theory of Mobile Processes.* Cambridge University Press, Cambridge, England, 2001.

[Gi14] M. Gieseking. *Refinement of $\pi$-Calculus processes.* Master thesis, Carl von Ossietzky Universität Oldenburg, 2014.

[En99] C. Ene, T. Muntean *Expressiveness of point-to-point versus broadcast communications.* In: FCT. Lecture Notes in Computer Science, vol 1684. Springer, Berlin, Heidelberg, 1999.

[Ol08] E. Olderog, H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification.* ISBN: 9781139474603, Cambridge University Press, 2008.

[Star] E. D'Osualdo. *Stargazer: A $\pi$-Calculus simulator.* http://www.emanueledosualdo.com/stargazer/

[ABC] S. Briais. *Another Bisimilarity Checker.* http://sbriais.free.fr/tools/abc/

[Ol18] E. Olderog. *Kombination von Spezifikationstechniken.* Vorlesungsskript, Carl von Ossietzky Universität Oldenburg, 2018/19.

[SIJ88] S. King and I. Sorensen and J. Woodcock. *Z: grammar and concrete and abstract syntax (Version 2.0).* Oxford University Computing Laboratory, England, 1988.

[TDC04] T. Kenji and D. Jin and C. Gabriel. *Relating $\pi$-Calculus to Object-Z.* Engineering of Complex Computer Systems, IEEE International Conference on 97-106, 2004.

[TDC02] T. Kenji and D. Jin. *An Overview of Mobile Object-Z.* 144-155. 10.1007/3-540-36103-0_17.

# Index

**R**

**S**



**T**



**U**

**V**

**W**

## Acknoldegment

Special thanks goes to:

- Prof. Dr. Ernst-Rüdiger Olderog for suggesting the topic of the thesis and for the continuous support.
- M.Sc Manuel Gieseking for providing the latex template.
- Dr. Emanuele D'Osualdo for support with $\pi$-calculus.

## Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 16. Juni 2020

______________________________

(Muhammad Ekbal Ahmad)