
Modelling Programming Languages for Concurrent and Distributed Systems in Specification Languages.

by

Chanchal Kumar Roy
Matr Nr.: 244644

Master Thesis

In the area of Computer Science

submitted to the
Faculty of Mathematics, Computer Science and Natural Sciences at the
RWTH Aachen University, Aachen, Germany, December 2004.

Supervised by

Priv.-Doz. Dr. Thomas Noll
Chair of Computer Science II

First Referee : Priv.-Doz. Dr. Thomas Noll
Chair of Computer Science II

Second Referee : Prof. Dr. Klaus Indermark
Chair of Computer Science II

Abstract

In this thesis work, a contribution to the field of *Formal Verification* is presented innovating a semantic-based approach for the verification of concurrent and distributed programs by applying model checking methods. Erlang is a declarative functional language for programming concurrent and distributed systems on which this thesis is aimed to employ the model checking methods. In contrast to the conventional approach of directly applying this verification technique to Erlang language, this thesis adopts the possibility of exploiting benefits from existing works by translating an Erlang program to a system model of the specification language π -calculus for which analysis and verification techniques have already been well established and there are existing tools for model checking π -calculus systems.

Dedication

To the departed souls of my parents & father-in-law.

Acknowledgements

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisor Priv.-Doz. Dr. Thomas Noll for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without him, this work would have been impossible. He has also unlocked the research potential within me.

I would like to thank Prof. Dr. Klaus Indermark to be my second referee and for spending his valuable time during my thesis presentation at Chair of Computer Science II on 25 November 2004.

I am very much grateful to my HIWI job supervisor Dipl.-Ing. Seoung-Hoon Oh for his cordial consideration and generous financial support during the whole period of my master study at Aachen.

I am also grateful to the RWTH Aachen University authority for awarding me merit-scholarship during my second year study that helped me much to concentrate more deeply in my thesis work.

I wish to thank my friends and classmates namely, Abu Kauser Jewel, Ismail Siraji, Mofizur Rahman, Nurjahan Nasrin and George Andonakis for their company and helping approach.

I express my appreciativeness to my family members and relatives especially, my mother-in-law Bela Rani Karmaker, youngest brother Ujjal Roy and sister-in-laws Kalyani Roy & Laboni Roy for their inspiration.

Finally, I am deeply indebted to my wife Banani Roy for more than I could ever express, due to her love, patience and support during this thesis work.

Table of Contents

1	Introduction.....	-1
1.1	Model Checking.....	-1
1.2	Motivation.....	-2
1.3	Related Work.....	-4
1.4	Overview.....	-5
2	Erlang.....	-7
2.1	Introduction.....	-7
2.2	Modules, Functions and Clauses.....	-8
2.3	Pattern Matching.....	-9
2.4	Data Objects.....	-9
2.5	Case and If.....	-10
2.6	Concurrency.....	-10
3	The Asynchronous π-calculus.....	-12
3.1	Introduction.....	-12
3.2	The Asynchronous π -calculus Syntax.....	-13
3.3	Free and Bound Occurrences of Names.....	-14
3.4	Structural Congruence and Reaction	-14
3.5	Scope Extrusion.....	-15
4	An Initial Approach to Translation Mapping.....	-17
4.1	Introduction.....	-18
4.1.1	Considerations.....	-18
4.1.2	General Notations.....	-20
4.1.3	Translation Functions.....	-20
4.1.4	Global Invariants.....	-21
4.1.5	PIErlang-00 Syntax	-21
4.2	Data Objects.....	-21
4.2.1	Integer	-21
4.2.2	Float.....	-24
4.2.3	Atom	-25
4.2.4	Variables.....	-26
4.3	Assignment Expressions.....	-27
4.3.1	Assignment Expression $X=E_1, E_2$	-27
4.3.2	Assignment Expression $X=E$	-28

4.4 Send Expression: PID as Implicit Mailbox	-29
4.5 Function Call Expressions.....	-31
4.5.1 n-ary Function Call.....	-31
4.5.2 n-ary Spawn Call.....	-32
4.6 Sequence of Expressions	-36
4.7 Receive Expression.....	-37
4.7.1 Matches of Receive Expression.....	-38
4.7.2 Match	-39
4.7.2(a) Match: Atoms as Patterns.....	-40
4.7.2(b) Match: Numbers as Patterns.....	-41
4.7.2(c) Match: Variables as Patterns.....	-42
4.8 Case Expression.....	-44
4.8.1 Case: As a Sequence of Two Expressions	-44
4.8.2 Case: As a Modified Receive Expression	-47
4.9. Receive and Case: Handling Non-determinism among Matches.....	-48
4.10 Function Definition.....	-52
4.11 PIERlang Program.....	-53
4.11.1 PIERlang Program 4.1.....	-54
4.11.1(a) Execution in Erlang Compiler.....	-54
4.11.1(b) Translation in the π -calculus.....	-55
4.11.1(c) The π -model	-57
4.11.1(d) Observing Behavior in the π -calculus.....	-57
4.11.2 PIERlang Program 4.2.....	-58
4.11.2(a) Execution in Erlang Compiler.....	-59
4.11.2(b) Translation in the π -calculus.....	-59
4.11.2(c) The π -model	-62
4.11.2(d) Observing Behavior in the π -calculus.....	-62
4.11.2(e) An Enriched FSM of Program 4.2.....	-63
4.12 TrPIs at a Glance.....	-65
4.12.1 Frequently Used TrPIs.....	-65
4.12.2 TrPIs for Handling Non-determinism among Matches.....	-67
5 Mapping Tuples with Polyadic Communications	-68
5.1 PIERlang-01 Syntax.....	-69
5.2 BIF self().....	-70
5.2.1 BIF self():Used As Argument.....	-71
5.2.2 PIERlang Program 5.1: self() as Argument.....	-71

5.2.2(a) Execution in PIERlang Compiler.....	-71
5.2.2(b) Translation in the π -calculus.....	-72
5.2.2(c) The π -Model.....	-72
5.2.2(d) Observing Behavior in π -calculus	-72
5.2.3 BIF self(): Used As Expression.....	-73
5.2.4 PIERlang Program 5.2: self() as Expression.....	-73
5.2.4(a) Execution in PIERlang Compiler.....	-73
5.2.4(b) Translation in the π -calculus.....	-74
5.2.4(c) The π -Model.....	-75
5.2.4(d) Observing Behavior in π -calculus	-75
5.3 Tuples as Expression.....	-76
5.4 Tuples in Send Expression: PID as Implicit Mailbox.....	-76
5.5 Tuples in Receive Expression: PID as Implicit Mailbox.....	-77
5.5.1 Receive Action in Polyadic π -calculus.....	-77
5.5.2 Matches of Receive Expression.....	-78
5.5.3 Matches	-79
5.5.3(a) Matches: Atoms & Numbers as Elements of Tuple	-80
5.5.3(b) Matches: Variables as Elements of Tuple.....	-80
5.5.4 PIERlang Program 5.3: CAR for DC Rule.....	-82
5.5.4(a) Execution in PIERlang Compiler.....	-82
5.5.4(b) Translation in the π -calculus.....	-83
5.5.4(c) The π -Model.....	-84
5.5.4(d) Observing Behavior in π -calculus: CAR for DC Rule.....	-85
5.5.5 PIERlang Program 5.4: Variables as Tuple Elements	-89
5.5.5(a) Execution in PIERlang Compiler.....	-89
5.5.5(b) Translation in the π -calculus.....	-90
5.5.5(c) The π -Model.....	-91
5.5.5(d) Observing Behavior in π -calculus.....	-91
5.5.6 PIERlang Program 5.5: Inaccuracy of Rule(26A).....	-92
5.5.6(a) Execution in PIERlang Compiler.....	-93
5.5.6(b) Translation in the π -calculus.....	-94
5.5.6(c) The π -Model.....	-95
5.5.6(d) Observing Behavior in π -calculus: Rule (26A) is Insufficient.	-95
5.5.7 An Approach to Improve Rule (26A).....	-96
5.5.7(a) Modification of Rule (26A): Providing BVS.....	-97
5.5.7(b) Modification of Rule (26B): Providing BNS.....	-98

5.5.7(c) Betterment of Rule (26C) over Rule(26B).....	-100
5.5.8 PIERlang Program 5.5: Rule (26C) is Sound but has Extra Names in BNS	-100
5.5.8(a) Execution in PIERlang Compiler.....	-100
5.5.8(b) Translation in the π -calculus.....	-100
5.5.8(c) The π -Model.....	-103
5.5.8(d) Observing Behavior in π -calculus.....	-104
5.5.9 Improving Rule (26C): Providing BNSRAV	-108
5.5.10 PIERlang Program 5.6: Rule (26D) Sounds Perfect.....	-109
5.5.10(a) Execution in PIERlang Compiler.....	-110
5.5.10(b) Translation in the π -calculus.....	-110
5.5.10(c) The π -Model.....	-113
5.5.10(d) Observing Behavior in π -calculus.....	-114
5.6 Tuples in Case Expressions.....	-119
5.6.1 PIERlang Program 5.7: Tuples in Case Expression	-119
5.6.1(a) Execution in PIERlang Compiler.....	-119
5.6.1(b) Translation in the π -calculus.....	-120
5.6.1(c) The π -Model.....	-121
5.6.1(d) Observing Behavior in π -calculus.....	-122
5.7 An Approach to Improve Send Rule (25).....	-123
5.7.1 PIERlang Program 5.8: Send rule(25) is Insufficient	-124
5.7.1(a) Execution in PIERlang Compiler.....	-124
5.7.1(b) Translation in the π -calculus.....	-125
5.7.1(c) The π -Model.....	-126
5.7.1(d) Observing Behavior in π -calculus.....	-126
5.7.2 A Modification to Send Rule (25).....	-127
5.7.3. PIERlang Program 5.8: Send Rule(25A) Sounds Correct.....	-127
5.7.3(a) Execution in PIERlang Compiler	-127
5.7.3(b) Translation in the π -calculus.....	-128
5.7.3(c) The π -Model.....	-128
5.7.3(d) Observing Behavior in π -calculus.....	-128
5.8 An Approach to Improve Tuple Expression Rule (24).....	-130
5.9 An Alternative Approach for Match Rule (26D).....	-130
5.10 TrPIs at a Glance.....	-131
6 Mapping Nested Tuples, Lists and Arithmetic Expressions.....	-132
6.1 PIERlang-02 Syntax.....	-132

6.2 Data Types.....	-133
6.3 Arithmetic Expressions.....	-134
6.4 Lists.....	-135
6.5 Nested Tuples.....	-135
6.6 Send Expression	-135
6.7 Matches	-136
6.8 PIERlang Program 6.1: A Different Approach	-136
6.8(a) Execution in PIERlang Compiler.....	-137
6.8(b) Translation in the π -calculus.....	-137
6.8(c) The π -Model.....	-139
6.8(d) Observing Behavior in π -calculus.....	-139
7 Mapping Guards.....	-142
7.1 PIERlang-03 Syntax.....	-142
7.2 Guards	-142
7.2.1 Guards in Function Definition.....	-143
7.2.2 Guards in Matches.....	-144
7.3 IF Expression	-144
8 Model Checking with HAL.....	-145
8.1 Introduction.....	-145
8.2 HAL compatible π -calculus	-146
8.3 HAL System Overview.....	-147
8.4 HAL Commands.....	-147
8.5 LTS from π -calculus Models.....	-148
8.5.1 LTS of Program 5.1.....	-148
8.5.2 LTS of Program 4.2.....	-149
8.5.3 LTS of Program 5.3.....	-150
8.5.4 LTS of Program 5.4.....	-150
9 Conclusion.....	-152
9.1 Future Works.....	-152
9.2 Summary.....	-154
References.....	-155

List of Figures

Figure 2.1 The Origins of Erlang.....	-7
Figure 3.1 The syntax of the asynchronous π -calculus.....	-13
Figure 3.2 Server S has access to printer P.....	-16
Figure 3.3 Client C has access to printer P.....	-16
Figure 4.1 PIERlang-00 syntax	-22
Figure 4.2 Graphical representation of rule(2)	-23
Figure 4.3 Graphical representation of $\text{res}'\langle\text{unknown}\rangle.\underline{\text{nil}} \parallel \text{res}(y).Q$	-23
Figure 4.4 Graphical representation of rule(4).....	-25
Figure 4.5 Graphical representation of $\text{TrPI}_{\text{exp}}(\text{self}, X)$	-26
Figure 4.6 Graphical representation of rule(7).....	-27
Figure 4.7 Graphical representation of rule(8).....	-28
Figure 4.8 Graphical representation of rule(9).....	-30
Figure 4.9 Graphical representation of rule(11).....	-33
Figure 4.10 Graphical representation of rule(12).....	-35
Figure 4.11 Graphical representation of rule(13).....	-36
Figure 4.12 Schematic diagram of the sender receiver Program 4.1.....	-55
Figure 4.13 Graphical representation of Simple FSM Program 4.2.....	-58
Figure 5.1: PIERlang-01 Syntax.....	-70
Figure 5.2 Schematic diagram of Program 5.3.....	-82
Figure 5.3 Schematic diagram of the echo process of Program 5.4.....	-89
Figure 5.4 Schematic diagram of the echo process of Program 5.5.....	-93
Figure 5.5 Communicating schematic diagram of locker Program 5.6.....	-110
Figure 5.6 Schematic diagram of the Program 5.7.....	-120
Figure 5.7 Schematic diagram of Program 5.8.....	-124
Figure 6.1 PIERlang-02 Syntax.....	-133
Figure 7.1 PIERlang-03 Syntax.....	-143
Figure 8.1 The π -calculus syntax compatible with HAL.....	-146
Figure 8.2 The logical architecture of HAL environment.....	-147
Figure 8.3 LTS of Program 5.1 (a) main process, (b) foo process.....	-148
Figure 8.4 LTS of Program 4.2 (a) start process, (b) $s4$ process (c) $s1/s2/s3$ process.....	-149
Figure 8.5 LTS of Program 5.3 (a) main/ping process, (b) pong process.....	-150
Figure 8.6 LTS of Program 5.4 (a) main process, (b) start process (c) loop process.....	-151

Chapter 1

Introduction

This chapter presents an introduction to this thesis. We start with the definition of *model checking* following a motivation behind this thesis work. Some related works are also mentioned along with an overview of the contents of this thesis paper.

Table of Contents \Rightarrow	1.1 Model Checking	-1
	1.2 Motivation	-2
	1.3 Related Work	-4
	1.4 Overview	-5

1.1. Model Checking

Formal verification means creating a mathematical model of a system, using a language to specify desired properties of the system in a concise and unambiguous way, and using a method of proof to verify that the specified properties are satisfied by the model. When the method of proof is carried out substantially by machine, we speak of *automatic verification*. Two well established methods to verification are theorem proving and model checking.

Model Checking[24], one of many formal verification methods, is an attractive and increasingly appealing alternative to simulation and testing to validate and verify systems. Given a system model and desired system properties, the *Model Checker* explores the full state space of the system model to check whether the given system properties are satisfied by the model. The Model Checker either verifies the given properties or generates counter examples.

While simulation and testing explore some of the possible behaviors of the systems, model checking conducts an exhaustive exploration of all possible behaviors. Thus, when the model checker verifies a given system property, it implies that all behaviors have been explored, and the question of adequate coverage or a missed behavior become irrelevant[24].

In this approach, the system to be verified is structured as a finite state transition system describing the behaviors of the system, and the specifications are expressed in a propositional temporal logic formula. Then, by exhaustively exploring the state space of the state transition system, it is possible to check automatically if the specifications are satisfied or to ascertain whether the finite state structure does actually represent a *model* for the formula. If the structure is indeed a model for the formula, then we can say that the system itself satisfies the property captured by the

formula. For specification languages such as Estelle, Lotos or SDL, *model checking* has been widely established as a verification technique by constructing Labelled Transition System(LTS). Once the LTS(s) is built, desirable system properties such as the absence of deadlocks can then be specified in a suitable logic like LTL or CTL, and can automatically be checked (at least for finite-state systems). The termination of model checking is guaranteed by the finiteness of the model.

There are two main advantages of using model checking compared to other formal verification methods:

- (1) It is fully automatic, and
- (2) It provides a counter example whenever the system fails to satisfy a given property.

1.2 Motivation

The main goal of this thesis is to provide a contribution to the field of *Formal Verification* by applying model checking methods to the Erlang language.

Erlang [1, 31] is a symbolic, high level and declarative programming language with support for concurrent and distributed programming. It has been developed at the Ericsson corporation and is typically used in telecommunication systems. It provides a functional sublanguage, enriched with constructs for dealing with side effects such as process creation and inter-process communication. Today many commercially available products offered by Ericsson are at least partly programmed in Erlang. The software of such products is typically organized into many, relatively small source modules, which at runtime execute as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test by manual methods. Automatic computer support is indispensable therefore. To fulfil this requirements, we provide an approach for automatic verification of Erlang programs by means of embedding model checking methods for Erlang language.

In contrast to the previous approach(like Estelle, Lotos and SDL above), however, labelled transition system(s) is not directly constructed for the given Erlang program, rather, this thesis innovates the idea of exploiting benefit from existing work by first translating the given Erlang program into a specification language for which analysis and verification methods have already been developed. Due to the dynamic and mobile communication structures which arise in many Erlang applications, classical

calculi such as CCS(Calculus of Communicating Systems) is not appropriate for this purpose. Instead, the π -calculus [34, 35] has been used, which was developed as a theoretical formalism for describing mobile systems and is best viewed as a formal framework for providing the underlying semantics of a high-level concurrent and/or distributed programming language.

The π -calculus is a name-passing calculus that allows the description of concurrent and distributed systems with dynamically changing interaction topology. Name communication, together with the possibility of declaring and exporting local names (scope extrusion), gives this calculus a great expressive power.

As Erlang is a concurrent and distributed programming language, processes in Erlang software tend to have a complicated communication structure. Erlang processes achieve concurrency through asynchronous message passing using Process Identifiers(PIDs) as the links of communication. This can be captured directly with the name passing feature of the asynchronous π -calculus. Erlang's message passing primitives have another promising feature: they can pass PIDs as messages to other process in communication. The π -calculus achieves this goal with the same idea: passing channels as messages in communication.

Indirect reference and dynamic creation of new process play a prominent role in interactions between processes in Erlang. For instance, one process can create a new child process and can send the PID of the newly created process to a second process by using asynchronous message passing. In Erlang, links serve as both communication channel and PID. Such interactions are typically hard to model using communicating finite state machine. The asynchronous π -calculus provides a simple way to model such interactions. In π -calculus, links are primitive names of communication channels. The combination of fresh name generation (new name creation) and private channel (fresh name) passing(scope extrusion) allows faithful modelling of several complicated communication patterns between software agents.

The above striking similarities between Erlang and π -calculus lead us to think about the possibility of performing Erlang verification tasks in π -calculus.

Once the π -calculus model is achieved for a given Erlang program, model checking methods can be applied with existing tools. One of the promising tools is HAL[9,10], exploits a novel automata-like model which allows finite state verification of systems specified in the π -calculus. The HAL system is able to interface with several efficient toolkits (e.g. model checkers) to determine whether or not certain properties hold for a

given specification. Another well known verification environment for π -calculus systems is the Mobility Workbench (MWB) [19, 21, 22] which started as a bisimulation equivalence checker. It contains everything needed for π -process analysis: π -grammar, parser, abstract representation, executor and bisimulation checker. The latest released version is MWB'99[21], which features a new faster model checker as well as decreased space requirements for open bisimilarity checking.

In [18], the possibility of verifying π -calculus processes via Promela[17, 37] translation is investigated. A general translation method from π -calculus processes to Promela models is presented and its usefulness is shown by performing verification tasks with translated π -calculus examples and SPIN[17, 30]. Model checking translated π -calculus processes in SPIN is shown to overcome shortcomings of the Mobility Workbench[19, 21, 22], which implements a theorem-proving style μ -calculus model checking algorithm for the π -calculus. In [16], a model checker for mobile systems specified in the style of the π -calculus is presented using tabled resolution.

As a final argue behind our this thesis work, it is noteworthy to mention that Microsoft Research has started a project called *Behave*[15, 12, 13, 14] in order to build tools for checking behavioral properties (like deadlock freedom, invariant checking, message understood properties) of asynchronous message-passing programs and their intention is to do this by directly analyzing source code written in an asynchronous programming language.

Thereby, in this thesis work, a translation from Erlang language to asynchronous π -calculus is provided with a view to have a system model in asynchronous π -calculus for a program written in Erlang. Additionally, it is observed that gained system model shows the same behavior as it could be expected from its corresponding Erlang programs.

1.3 Related Works

The attempt to use the π -calculus for verifying Erlang programs should be seen as one among several similar projects. Similar approaches with different tools and languages have been used by others.

In [11], a translation of Erlang into Promela and checking Promela by SPIN[17, 30] was experimented. The main outcome of this experiment was that Promela (and hence SPIN) is too far away from Erlang to really use them together.

In [7], a translation from Erlang to μCRL is presented. The language μCRL [8] is a process algebra with data and several verification tools[5, 6] are available for μCRL , including a tool to create labelled transition systems from μCRL specifications. By having a translation from Erlang to μCRL , the verification tools for process algebras and labelled transition systems can be applied to industrial code.

In [29], the Java PathFinder, *JPF*, a translator from a subset of Java 1.0 to Promela, the programming language of the SPIN[17, 30] model checker is presented. The purpose of JPF is to establish a framework for verification and debugging of Java programs based on model checking. The system is especially suited for analyzing multi-threaded Java applications, where normal testing usually falls short. The system can find deadlocks and violations of boolean assertions stated by the programmer in a special assertion language.

1.4 Overview

In Chapter 1, a motivation behind this thesis along with some related works has been presented.

In Chapter 2, an introduction to the Erlang language is provided. Apart from briefly describing how to write simple sequential programs, we focus on the language's concurrency and inter-process communication features.

In Chapter 3, a brief presentation of the syntax and semantics of the asynchronous π -calculus is given.

In Chapter 4, we introduce a restricted subset of Erlang language named *PIErlang-00* and for each of the syntactic constructs of *PIErlang-00*, a corresponding translation mapping in monadic asynchronous π -calculus is provided. Finally, π -calculus system models of two complete *PIErlang-00* programs have been built based on the translation mapping rules of the syntactic constructs. Moreover, it has been shown that gained π -calculus models are showing the same behavior as it could be expected from its corresponding *PIErlang-00* programs.

In Chapter 5, *PIErlang-00* is enriched with *tuples* and a BIF *self()*. We call this version *PIErlang-01*. A detailed step by step translation for tuple-based communication in the form of the polyadic asynchronous π -calculus with several examples has been presented. Additionally, the behaviors of the π -models has been observed and found that they are showing exactly the same behaviors as its corresponding Erlang programs although in some cases there is likely a possibility of having a deadlock.

Besides this, a modification to the semantics of π -calculus has been proposed to keep the translation procedure easier.

In Chapter 6, PIERlang-02 is formed from PIERlang-01 by supporting nested tuples, lists and arithmetic expressions. As usual, a translation mapping for PIERlang-02 has been provided. Additionally, one PIERlang-02 program has been used to get corresponding system model in π -calculus by applying translation mapping rules. Furthermore, it has been shown that gained π -calculus model shows the same behavior as it could be expected from its corresponding PIERlang-02 program.

In Chapter 7, Guards, one of the interesting and promising feature of Erlang language, are added to PIERlang-02, PIERlang-02 renamed as PIERlang-03 and translation mapping supporting guards are presented.

In Chapter 8, the verification tool HAL is introduced with a view to provide an idea of how to verify the gained π -models in HAL.

In Chapter 9, a summary of the works that we have done is given and some hints for future works are made.

Chapter 2

Erlang

This chapter introduces Erlang. The treatment of the language is not intended to be complete. For fuller treatment the reader is referred to [1]. Developments to Erlang since [1] can be found in the OTP documentation [31]. A more formal treatment of Erlang can be found in the Erlang reference manual[32] and in the core Erlang specification [26]. To concentrate on the real-time aspects of the language, the reader is referred to [27]. The text of this chapter has been taken from [25].

	2.1 Introduction	-7
	2.2 Modules, Functions and Clauses	-8
	2.3 Pattern Matching	-9
Table of Contents ⇒	2.4 Data Objects	-9
	2.5 Case and If	-10
	2.6 Concurrency	-10

2.1 Introduction

Erlang is a Programming language which is designed for programming concurrent, real-time, distributed fault-tolerant systems. The Erlang programming language has been developed at Ericsson and Ellemtel Computer Science Laboratories.

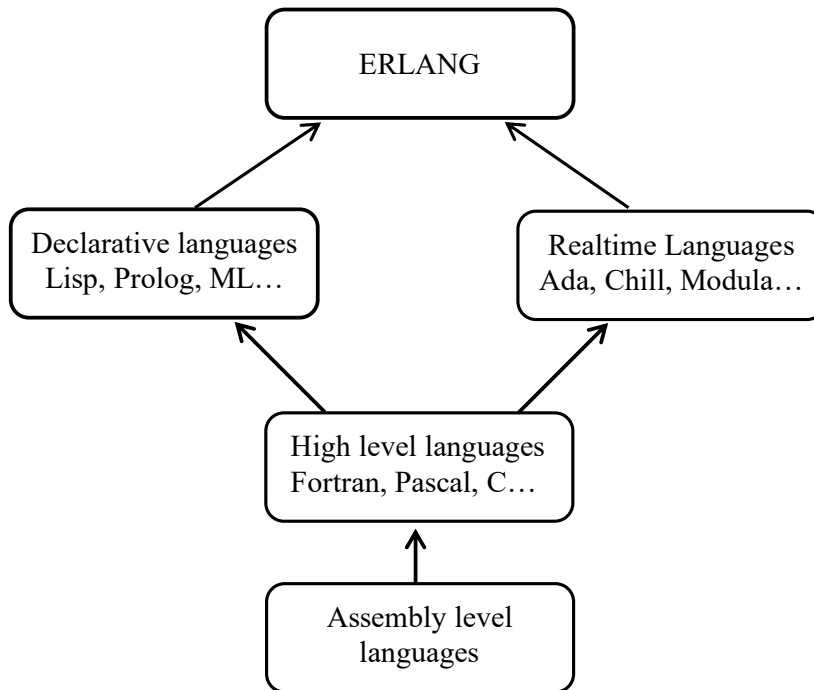


Figure 2.1 The Origins of Erlang

Development started in the early eighties with experiments of programming telecom using different languages. The experiments showed that no existing language had all of the following features:

- high level declarative language
- primitives for concurrency and error recovery
- concurrency

Therefore, a new language with these features was developed, Erlang. In the beginning of the nineties the first implementation of Erlang was released to users. Today Erlang is available for most operating systems and platforms and is developed, maintained and marketed by Erlang Systems Division.

Erlang is a functional concurrent language designed for implementing reliable real-time systems. It is a small but powerful language which a person with some knowledge of programming can learn in a few days time.

2.2 Modules, Functions and Clauses

An Erlang program is divided into modules. A module consists of one or several functions which in turn consists of clauses. The functions in a module are hidden from outside the module except for the exported functions which can be called from outside.

```
-module(mathlib).
-export(factorial/1).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

Program 2.1 A simple factorial program

The module *mathlib* above consists of one function, *factorial/1*, which means the function has an arity (number of arguments) of 1. Factorial consists of two clauses separated by a semicolon and ended by a full stop. The function factorial is exported and can be called from outside the module. Functions that are not exported are local and can only be called from inside the module. An external call to factorial/1 has the following syntax:

```
mathlib: factorial(3).
```

This call would result in the answer 6.

It is also possible to import functions so that they can be used as if they were local functions. This is done by using *import*.

Erlang also supports last call optimization. This allows functions to be evaluated in constant space i.e. the stack does not grow for each recursive call.

2.3 Pattern Matching

When a function is called one of its clauses is chosen. Erlang chooses its clauses by pattern matching which is a fundamental concept which contributes in making Erlang programs short and succinct. This means that the first clause which matches the pattern will be chosen. In the example call above (Program 2.1) *factorial(0)* will only be matched when the argument is 0, otherwise the second clause will be chosen. When a match occurs the expression on the right hand side of the “->” will be evaluated. Pattern matching is also used for assigning variables. Variables in Erlang are only assigned once and cannot change value in the same scope. They are also untyped and are bound to a type when they are bound to a value. In the factorial example (Program 2.1) N is first bound to 3 and then factorial is called again with its N bound to 2 and so on... This means that a scope of a variable is from it was bound to the end of the clause it was bound in.

```
> X={4711, foobar}.
    {4711, foobar}
> {A, B}=X
    {4711, foobar}
> A.
    4711
> B.
    foobar
```

2.4 Data Objects

An object in Erlang is either a constant, compound term or a variable. A constant can be an atom, float, process identifier(PID) or an integer while a compound term is either a list or a tuple. A list is a construct of pairs in the form [H | T] or it is an empty list[], Thus [a, b, c] is short for [a | [b | [c | []]]]. Strings are also lists; “abc” is short for [97, 98, 99]. The tuple {a, b, c} is a tuple of size 3 with atoms as elements. Lists are used when the number of elements change dynamically while tuples are used when the number of elements are static. Variables begin with an upper case letter and “_” denotes anonymous variable. Some examples of atoms, lists, and tuples are given below:

atoms: foo, 'hey you' , hello_world

lists: [a, b, c] , [42, foo], []

tuples: {a, b, c}, {1, foobar}, {}

2.5 Case and If

The *case* and *if* expressions are used for choice between alternatives within the body of a clause.

```

case Expr of
    Pat1 [when Guard1] -> Seq1;
    Pat2 [when Guard2] -> Seq2;
    ...
    PatN [when GuardN] -> SeqN
end

if
    Guard1 -> Seq1 ;
    Guard2 -> Seq2;
    ...
end

```

In the *case* expression, the *Expr* is first evaluated and then the pattern which fits the evaluation is chosen and the corresponding sequence is run. *When* is a guard test which also can be used in the head of a function.

The *if* expression, on the other hand, has its guards sequentially evaluated and the first guard that succeeds has its sequence evaluated.

In both *if* and *case*, it is an error if no pattern matches. Therefore a “catch all” clause should be added.

2.6 Concurrency

Erlang has a number of multi-process primitives. *Spawn/3* is a primitive which spawns a new concurrent process. *Spawn/3* has the following syntax:

```
ProcessID = spawn(Module, Function, [Arg1, Arg2, ...,])
```

spawn returns the process identifier(PID) of the new process to variable *ProcessID*. Processes communicate with each other by sending messages. This is done by the send primitive “!”.

ProcessID ! Message

Messages which can be any Erlang term sent by a process are received by the primitive *receive*. A *receive* expression will suspend the process until a message that will match any of the patterns has arrived. The messages in a process mailbox are matched in FIFO order. If a message does not match it is left in the mailbox until a match for that message is found and the next message is checked.

```
receive
```

```
    Message1 [when Guard1] -> Action1;
```

```
    ...
```

```
    MessageN [when GuardN] -> ActionN
```

```
[after
```

```
    Time -> ActionTimeOut]
```

```
end
```

Time is an optional time out expression which timeouts after *Time* milliseconds.

```
-module(talk).
```

```
-export([start/0, say_hello/1]).
```

```
start() ->
```

```
    ProcessID = spawn(talk, say_hello, [self()]),
```

```
    ProcessID ! {self(), hello},
```

```
    say_hello(ProcessID).
```

```
say_hello(ProcessID) ->
```

```
    receive
```

```
        { ProcessID, hello} ->
```

```
            ProcessID ! {self(), hello}
```

```
    end,
```

```
    say_hello(ProcessID).
```

Program 2.2 A simple echo process.

A process which runs the module *talk* will spawn a process at which it will send and receive messages from. After the child process is spawned, the parent process will send a message containing its *PID* and the atom *hello* to its child. The child process receives the message and sends an answer at which the parent receives and answers and so on... *self()* is a BIF(Built In Function) which returns the processes own PIDs.

Chapter 3

The Asynchronous π -calculus

In this chapter, a short introduction to the asynchronous π -calculus is given. The asynchronous π -calculus is a variant of the π -calculus[3, 34, 35, 36] based on the idea that the messages are elementary processes that can be sent without any sequencing constraint. This chapter only focuses on the syntax and structural congruence and reaction relation of the calculus. A detailed study on asynchronous π -calculus can be found in [4, 33].

	3.1 Introduction	-12
	3.2 The Asynchronous π -calculus Syntax	-13
Table of Contents \Rightarrow	3.3 Free and Bound Occurrences of Names	-14
	3.4 Structural Congruence and Reaction	-14
	3.5 Scope Extrusion	-15

3.1 Introduction

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

The π -calculus is available in two basic styles: the monadic π -calculus, where exactly one name is communicated at each synchronization, and the polyadic π -calculus, where zero or more names are communicated. We will use both monadic and polyadic asynchronous π -calculus together as mentioned in Figure 3.1. below. The basic concept behind the π -calculus is naming or reference. Names are the primary entities and they may refer to links or channel or any other kind of basic entity. Processes, sometimes referred as agents, are the other kind of entities. We let the letters X_1, \dots, X_n , x, y, y_1, \dots, y_n range over the names. We also let π, π_1, π_2 range over agents or process expressions.

System:	$S ::= Q^+$	
Process Definition:	$Q ::= A(x_1, \dots, x_n) = \pi$ (where $i \neq j \Rightarrow x_i \neq x_j$)	$; n \geq 0$
Process:	$\pi ::= \underline{nil}$	Nil
	$ \alpha . \pi$	Prefix
	$ x' < y_1, \dots, y_n > . \underline{nil}$	Asynchronous Output ; $n \geq 0$
	$ \pi_1 \pi_2$	Parallel
	$ \pi_1 + \pi_2$	Sum
	$ (\text{new } x) \pi$	Restriction
	$ [x=y] \pi$	Match
	$ [x \neq y] \pi$	Mismatch
	$ A(x_1, \dots, x_n)$	Identifier/Process Instantiation $n \geq 0$
Action Prefixes:	$\alpha ::= x(y_1, \dots, y_n)$	Input
	$ \tau$	Silent

Figure 3.1 The syntax of the asynchronous π -calculus.

3.2 The Asynchronous π -calculus Syntax

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

- (1) nil is an inactive or deadlock process; it is a process that can do nothing.
- (2) The prefix $\alpha . \pi$ has a single capability, expressed by α ; the process π cannot proceed until that capability has been exercised. The prefixes could be in two possible forms $x(y). \pi$ or $\tau . \pi$. “ $x(y)$ ” is a positive prefix, where x is the input port of an agent; it binds the variable y . At port x the arbitrary name z is input by $x(y). \pi$, which behaves like $\pi[y/z]$, where $\pi[y/z]$ is the result of substituting z for all free (unbound) occurrences of y in π . Similarly, arbitrary names z_1, \dots, z_n are input by $x(y_1, \dots, y_n). \pi$ at port x and behaves like $\pi[y_1/z_1, \dots, y_n/z_n]$. τ is the silent action; $\tau . \pi$ first performs the silent action and then acts like π .
- (3) $x' < y > . \underline{nil}$ is an asynchronous output process. “ $x' < y >$ ” is a negative prefix; x' can be thought of an output port of an agent which contains it. $x' < y > . \underline{nil}$ outputs y on port x then behaves like \underline{nil} that can do nothing. We are using asynchronous π -calculus and therefore, only \underline{nil} could follow an output action. Similarly, $x' < y_1, \dots, y_n > . \underline{nil}$ outputs y_1, \dots, y_n on port x then behaves like \underline{nil} .

- (4) A parallel composition $\pi_1 \parallel \pi_2$, which represents the combined behavior of π_1 and π_2 executing in parallel. The components of π_1 and π_2 can act independently, and may also communicate if one performs an output and the other an input along the same port.
- (5) $\pi_1 + \pi_2$ represents sum-nondeterminism, that is, do either process π_1 or process π_2 .
- (6) $(\text{new } x) \pi$ means that x is declared as a *new name* local to process π and is bound in π . It is not visible outside π .
- (7) $[x=y]\pi$ represents the process that changes to π if $x=y$. Mismatch is the opposite, i.e., it checks $x \neq y$.
- (8) $A(x_1, \dots, x_n)$ represents the instantiation of a defined agent.
- (9) $A(x_1, \dots, x_n) = \pi$ (where $i \neq j \Rightarrow x_i \neq x_j$) represents the declaration of a process A in terms of process π . One can think of it as a procedure declaration in traditional procedural programming.

3.3 Free and Bound Occurrences of Names

The *input prefix* and the *new operator* bind the names. For example, in a process $x(y).$ π , the name y is bound. In $(\text{new } x)\pi$, x is considered to be bound. Every other occurrences of a name like x in $x(y).\pi$ and x, y in $x' \langle y \rangle.\pi$ are free.

3.4 Structural Congruence and Reaction

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

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

- (i) Renaming of bound names i.e. α -conversion.
- (ii) Reordering of terms in a summation i.e., commutativity of “+” i.e.,

$$P + Q \equiv Q + P, \quad P + (Q + R) \equiv (P + Q) + R$$
- (iii) $P \parallel Q \equiv Q \parallel P, \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R, \quad P \parallel \underline{\text{nil}} \equiv P$
- (iv) $(\text{new } x) (P \parallel Q) \equiv P \parallel (\text{new } x) Q$ if $x \notin \text{freeName}(P)$, $(\text{new } x) \underline{\text{nil}} \equiv \underline{\text{nil}}$

The structural congruence is a congruence relation on P^π i.e. if $P \equiv Q$ then,

- (i) $\pi.P + R \equiv \pi.Q + R$
- (ii) $P \parallel R \equiv Q \parallel R$ and $R \parallel P \equiv R \parallel Q$
- (iii) $(\text{new } x) P \equiv (\text{new } x) Q$

The reaction relation $\Rightarrow \subseteq P^\pi$ is generated by the following rules:

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

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

$$\frac{P \Rightarrow P'}{P \parallel Q \Rightarrow P' \parallel Q} (\text{par})$$

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

$$\frac{P \Rightarrow P'}{Q \Rightarrow Q'} (\text{struct}) \text{ if } P \equiv Q \text{ and } P' \equiv Q'$$

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

3.5 Scope Extrusion

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

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

Initially (Figure 3.2), only server process S has access to printer process P using a private link a . We suppose that S also has a link b to the client process C and using link b , client C can request server S to get access to printer P .

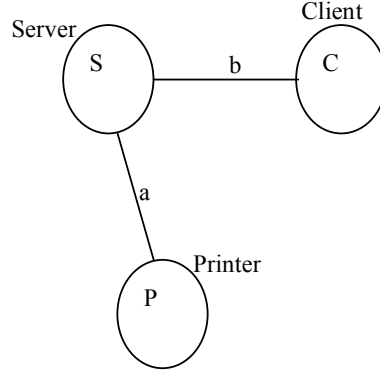


Figure 3.2 Server S has access to printer P

As per the request of the printer P , server S is now willing to pass link a (private to S and P) to C . Let $S = b' \langle a \rangle . S'$ and $C = b(y) . C'$. Then the composite process is represented by

$$(\text{new } a) (b' \langle a \rangle . S' \parallel P) \parallel b(y) . C'$$

Here *new* a represents the privacy of S 's link to P . After a communication along b from S to C the process becomes

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

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

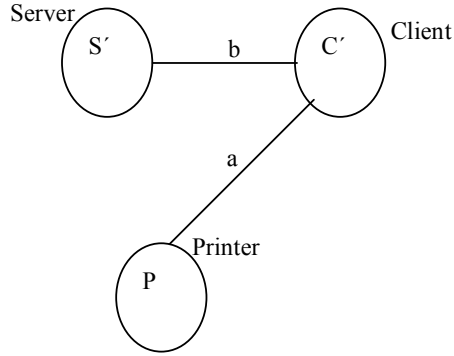


Figure 3.3 Client C has access to printer P

Chapter 4

An Initial Approach to Translation Mapping

In this chapter, the issue of a formal translation from a subset of Erlang to the π -calculus has been presented. Some considerations, general notations and global invariants are also provided to have the translation procedure easier. After presenting the translation rules for each of the syntactic constructs of the Erlang subset, two complete Erlang programs are translated to the π -calculus models. Furthermore, the behaviors of gained π -models are compared with the behaviors of their corresponding Erlang programs.

	4.1 Introduction	-18
	4.1.1 Considerations	-18
	4.1.2 General Notations	-20
	4.1.3 Translation Functions	-20
	4.1.4 Global Invariants	-21
	4.1.5 PIERlang-00 Syntax	-21
	4.2 Data Objects	-21
	4.2.1 Integer	-21
	4.2.2 Float	-24
	4.2.3 Atom	-25
	4.2.4 Variables	-26
	4.3 Assignment Expressions	-27
Table of Contents \Rightarrow	4.3.1 Assignment Expression $X=E_1, E_2$	-27
	4.3.2 Assignment Expression $X=E$	-28
	4.4 Send Expression: PID as Implicit Mailbox	-29
	4.5 Function Call Expressions	-31
	4.5.1 n-ary Function Call	-31
	4.5.2 n-ary Spawn Call	-32
	4.6 Sequence of Expressions	-36
	4.7 Receive Expression	-37
	4.7.1 Matches of Receive Expression	-38
	4.7.2 Match	-39
	4.7.2(a) Match: Atoms as Patterns	-40
	4.7.2(b) Match: Numbers as Patterns	-41
	4.7.2(c) Match: Variables as Patterns	-42
	4.8 Case Expression	-44
	4.8.1 Case: As a Sequence of Two Expressions	-44
	4.8.2 Case: As a Modified Receive Expression	-47

4.9. Receive and Case: Handling Non-determinism among Matches	-48
4.10 Function Definition	-52
4.11 PIERlang Program	-53
4.11.1 PIERlang Program 4.1	-54
4.11.1(a) Execution in Erlang Compiler	-54
4.11.1(b) Translation in the π -calculus	-55
4.11.1(c) The π -model	-57
4.11.1(d) Observing Behavior in the π -calculus	-57
4.11.2 PIERlang Program 4.2	-58
4.11.2(a) Execution in Erlang Compiler	-59
4.11.2(b) Translation in the π -calculus	-59
4.11.2(c) The π -model	-62
4.11.2(d) Observing Behavior in the π -calculus	-62
4.11.2(e) An Enriched FSM of Program 4.2	-63
4.12 TrPIs at a Glance	-65
4.12.1 Frequently Used TrPIs	-65
4.12.2 TrPIs for Handling Non-determinism among Matches	-67

Table of Contents \Rightarrow

4.1 Introduction

This chapter develops the translation procedure in a number of steps to illustrate the decisions leading to the final design for a complete Erlang Program. Although in Chapter 2, we have presented Erlang syntactic constructs altogether, during translation mapping we have used four different Erlang versions starting from a very restricted one and then gradually added more constructs to have a wider version of Erlang with possible translation mapping in the asynchronous π -calculus.

4.1.1 Considerations

Erlang is a full programming language. It is not the aim to represent the behavior of a given Erlang system in all of its facets. Rather, we have abstracted from certain details to simplify the translation. Thus, initially only the following aspects will be covered:

- Process creation,
- Sending and receiving of messages,
- Function calls,
- Sequential control flow.

The following aspects related to data structures are ignored, although in some cases, certain parts of the data space such as atoms are considered to be modeled.

- Computations on low-level data types such as numbers,
- Pattern matching,
- Deterministic branching (in *if/case/receive* expressions).

As per the asynchronous message passing in Erlang, two ways of handling the potentially unbounded mailbox are conceivable.

- Using Synchronous Message Passing.
- Using Asynchronous Message Passing.

Erlang uses asynchronous message passing scheme. Therefore, if we want to use synchronous message passing, we have to model the mailbox explicitly. While modelling mailbox explicitly, we can bound the size of the mailbox or we can support the full semantics (unbounded size).

We have found that asynchronous π -calculus is one of the useful variants of π -calculus to model asynchronous message passing scheme. But if we use asynchronous π -calculus for Erlang's asynchronous message passing, order of messages within the sequential process is not respected which violates the semantics of mailbox in Erlang. However, we also found that modelling mailbox explicitly with synchronous message passing is more complicated and less useful than directly modelling asynchronous message passing although order of messages is not respected. Thereby, we will use asynchronous π -calculus for translation mapping by modelling the mailbox implicitly.

Furthermore, the translation mapping definitions are built on the formal syntax of the presented Erlang programs with the following simplest settings:

- No complex data structures.
- No timing restrictions.

Altogether, the mapping from Erlang to the asynchronous π -calculus is provided with several parameters which allow to tune the translation with respect to the above aspects.

4.1.2 General Notations

In this section, some general notations are introduced. One of them is *PID* which will be used as an acronym for *Process Identifier*.

We use the word π -calculus to mean the *asynchronous (polyadic + monadic) π -calculus*. In this chapter, only *monadic asynchronous π -calculus* is used for translation mapping.

While translating Erlang to π -calculus or derivation of a rule with a basic rule we have used the notation:

$$\textbf{(n)} \\ \textbf{Exp1} = \textbf{Exp2}$$

Here n corresponds to a rule number which has been applied to derive expression *Exp2* from *Exp1*. Some times more than one rules have been applied to get the *Exp2* from *Exp1* skipping some intermediate steps.

We have used the notation, $\textbf{(n1)} \rightarrow \textbf{(n2)}$ to explicitly mention that first *rule (n1)*

$$\textbf{Exp1} = \textbf{Exp2}$$

has to be applied on *Exp1* and then on the intermediate expression *rule (n2)* has been applied to get expression *Exp2*.

While observing the translated π -calculus system behaviors, we have used the notation,

$$\textbf{(string)} \\ \textbf{System1} \Rightarrow \textbf{System2}$$

Where *System2* state has been achieved from *System1* state by applying the π -calculus reaction rule or process definition and it will be indicated by *string*. Most of the cases, we have used π -calculus *Structural Congruence*(cf. Chapter 3) among process expressions to simplify the definition of reaction relation but we have neither mentioned explicitly in *string* nor showed the intermediate system state(s).

4.1.3 Translation Functions

Two frequently used functions for translation mapping are TrPI_{arg} and TrPI_{exp} . TrPI_{arg} is used to translate any *Argument* of Erlang. Its signature is give below:

$\text{TrPI}_{\text{arg}}: \text{Argument} \rightarrow \text{Name}$

This signature indicates that TrPI_{arg} can translate any *Number*, *Atom* or *Variable*(cf. Figure 4.1) to a *Name* in the π -calculus when they are used as the arguments of *function calls*, *spawn calls* and *send* expressions. This is also needed while working with patterns of *receive* expressions.

TrPI_{exp} is used to translate any expression of Erlang to π -calculus. It has the following signature:

TrPI_{exp} : Name X Expression \rightarrow Process.

This signature indicates that TrPI_{exp} will take a Name (normally PID for the corresponding Expression) and an Erlang expression as input and will produce a process as output. We will discuss both of the functions in the subsequent sections with examples.

Some other functions like $\text{TrPI}_{\text{match}}$, $\text{TrPI}_{\text{fundef}}$ and $\text{TrPI}_{\text{program}}$ are used during the translation procedure. Each of them will be discussed in details in the corresponding sections.

4.1.4 Global Invariants

We have considered some global invariants during translation. These are as follows:

When the evaluation is terminated, the process $\text{TrPI}_{\text{exp}}(\text{self}, E)$ sends the value of the expression E along some distinguishable channel. Normally, channel *res* will be used for this purpose.

The evaluation result of a function *fun* will be sent along *fun_res* channel.

Name *dummy* will be used frequently to receive evaluation result of expression and then will be discarded (later).

4.1.5 PIERlang-00 Syntax

In this chapter, a restricted subset of Erlang is chosen for translation mapping. We call this subset PIERlang-00. The syntax of PIERlang-00 is shown in Figure 4.1. We will discuss each of the syntactic constructs of PIERlang-00 while translating to π -calculus.

4.2 Data Objects

In PIERlang-00 syntax (cf. Figure 4.1), we have considered 2 simple data types, Numbers(Integer & Float) and Atom. Here we have tried to have a corresponding mapping in the π -calculus for each of them along with Variables.

4.2.1 Integer

As in π -calculus there are only *Names* and as our intention is to translate any Erlang integer number into π -calculus, we have found that an *integer number* of Erlang could

Program	$P ::= F+ ; E$	
Function Definition	$F ::= f(X_1, X_2, \dots, X_n) \rightarrow E$	$; n \geq 0$
Expression	$E ::= n \mid a \mid X$	
	$\mid X = E_1, E_2 \mid X = E \mid E_1, E_2$	
	$\mid f(A_1, A_2 \dots A_n) \mid$	$; n \geq 0$
	$\mid \text{spawn}(f, [A_1, A_2, \dots, A_n])$	$; n \geq 0$
	$\mid A_1 ! A_2$	
	$\mid \text{receive } M_1; \dots; M_n \text{ end}$	$; n > 0$
	$\mid \text{case } E \text{ of } M_1; \dots; M_n \text{ end}$	$; n > 0$
Match	$M ::= P \rightarrow E$	
Pattern	$P ::= n \mid a \mid X$	
Argument	$A ::= n \mid a \mid X$	
$n \in \text{Numbers (Integer \& Float)};$		
$a, f \in \text{Atoms};$		
$X, X_1, \dots, X_n \in \text{Variables}$		

Figure 4.1 PIERlang-00 syntax

be translated to a *Name* in π -calculus. We have not considered to represent any constant in π -calculus semantically and therefore, we have decided that an Integer number should be an unknown name in the π -calculus.

In our PIERlang, an integer number could be used as an argument or as an expression and thereby, there are two kinds of mapping in the π -calculus for any integer number.

We have considered that if any integer is used as the Argument (cf. Figure 4.1) then there should be a direct mapping in the π -calculus, the unknown name which can be formally written as:

$$\text{TrPI}_{\text{arg}}(n) := \text{unknown} \quad -(1)$$

Some examples:

$$\text{TrPI}_{\text{arg}}(10888) = \text{unknown}$$

$$\text{TrPI}_{\text{arg}}(-10888) = \text{unknown}$$

$$\text{TrPI}_{\text{arg}}(\$A) = \text{unknown}$$

$$\text{TrPI}_{\text{arg}}(1\#0888) = \text{unknown}$$

$$\text{TrPI}_{\text{arg}}(16\#1A) = \text{unknown}$$

From rule (1) along with the examples above, its clear that TrPI_{arg} takes any number as input and produces *unknown*, a name in the π -calculus as output.

We have also considered that if any integer is used as an expression, then there is most likely that it will be used by the subsequent expression(s). Having this in mind, we have introduced a new global name *res* in π -calculus to store the results of such atomic expression. It is done by sending the integer number as the argument of the *send* expression along the *res* channel. This can be formulated as follows:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, n) &:= \text{res}'\langle \text{TrPI}_{\text{arg}}(n) \rangle.\underline{\text{nil}} \\ (1) \quad &= \text{res}'\langle \text{unknown} \rangle.\underline{\text{nil}} \end{aligned} \quad -(2)$$

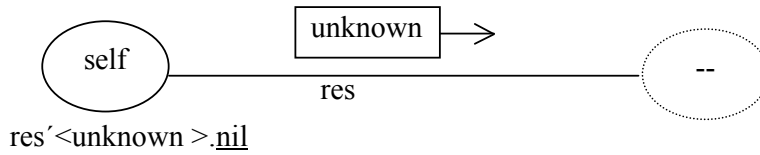


Figure 4.2 Graphical representation of rule(2)

As integer number n is used here (rule(2)) as the argument of the *send* expression, rule(1) is used to get its corresponding translation.

From the above representation, it is clear that any process can receive *unknown* (here π -calculus corresponding name of any integer number) with the *receive* expression of π -calculus along the channel *res* by executing in parallel with the *send* expression above. Let consider there is a process $\text{res}(y).Q$ (with PID *qpid*) which is ready to receive any name z on port *res* and then would behave like $Q[y/z]$. It can be represented formally as follows:

$$\text{res}'\langle \text{unknown} \rangle.\underline{\text{nil}} \parallel \text{res}(y).Q \xrightarrow{(\text{react})} \underline{\text{nil}} \parallel Q[y/\text{unknown}]$$

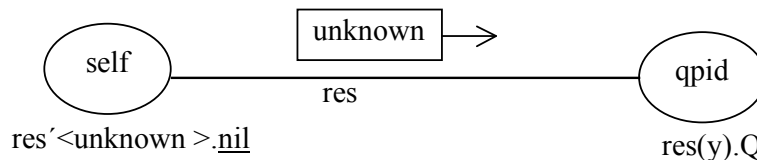


Figure 4.3 Graphical representation of $\text{res}'\langle \text{unknown} \rangle.\underline{\text{nil}} \parallel \text{res}(y).Q$

In this way, every free occurrence of y in Q will be replaced by *unknown*.

In rule (2), another global name *self* is included as the first parameter of the translation function. Every expression in Erlang is evaluated by a certain process and normally this is represented in Erlang with a BIF *self()*, the current process executing the expression.

Some examples are:

```
TrPIexp(self, 10888) := res'<unknown>.nil
TrPIexp(self, -10888) := res'<unknown>.nil
TrPIexp(self, $A) := res'<unknown>.nil
TrPIexp(self, 1#0888) := res'<unknown>.nil
TrPIexp(self, 16#1A) := res'<unknown>.nil
```

4.2.2 Float

We have also found that for any floating number in Erlang, there should be a corresponding name in π -calculus for the same reason described above in the case of Integer number. Therefore,

$$\text{TrPI}_{\text{arg}}(\text{fl}) := \text{unknown} \quad \text{-(1A)}$$

Some examples are:

```
TrPIarg(16.0)=unknown
TrPIarg(-10.33)=unknown
TrPIarg(-1.8e2)=unknown
TrPIarg(-0.36e-2)=unknown
TrPIarg(1.0e6)=unknown etc.
```

For floating point expressions, we have used the same formalism as Integer expressions as follows:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{fl}) &:= \text{res}'<\text{TrPI}_{\text{arg}}(\text{fl})>.\underline{\text{nil}} \\ &\quad (1A) \\ &= \text{res}'<\text{unknown}>.\underline{\text{nil}} \end{aligned} \quad \text{-(2A)}$$

Some examples are:

```
TrPIexp(self, 16.0)= res'<unknown>.nil
TrPIexp(self, -10.33)= res'<unknown>.nil
TrPIexp(self, -1.8e2)= res'<unknown>.nil
TrPIexp(self, -0.36e-2)= res'<unknown>.nil
TrPIexp(self, 1.0e6)= res'<unknown>.nil
```

However, from the translation mapping of Integer and Floating point numbers, it has been found that both mappings can be treated in the same way and therefore, from now for any number in PIERlang, rules (1) and (2) will be used ignoring (1A) & (2A).

4.2.3 Atom

Atoms are constant names in Erlang. The value of an atom is its name. Two atoms are equivalent when they are spelt identically. We have found an easy mapping for atoms in π -calculus, the so called names. We have taken the proposition that any atom in PIERlang can be translated to a new global name in the π -calculus with the same spelling and context of that atom without any changes.

However, depending on the context where an atom is used, the translation has been divided in two types like for numbers mentioned above.

$$\text{TrPI}_{\text{arg}}(a) := a \quad \text{-(3)}$$

and

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, a) &:= \text{res}'\langle \text{TrPI}_{\text{arg}}(a) \rangle.\underline{\text{nil}} \\ (3) \quad &= \text{res}'\langle a \rangle.\underline{\text{nil}} \end{aligned} \quad \text{-(4)}$$

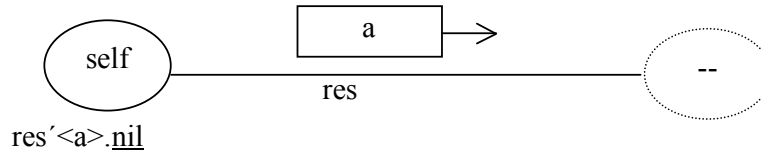


Figure 4.4 Graphical representation of rule(4)

Some examples are:

Using rule (3)

$$\text{TrPI}_{\text{arg}}(\text{start}) := \text{start}$$

$$\text{TrPI}_{\text{arg}}(\text{end}) := \text{end}$$

$$\text{TrPI}_{\text{arg}}(\text{start_end}) := \text{start_end}$$

Using rule (4)

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{start}) := \text{res}'\langle \text{start} \rangle.\underline{\text{nil}}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{end}) := \text{res}'\langle \text{end} \rangle.\underline{\text{nil}}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{start_end}) := \text{res}'\langle \text{start_end} \rangle.\underline{\text{nil}}$$

4.2.4 Variables

Erlang variables cannot be typed. A variable can be bound to any term. The scope (region of the program in which a variable can be accessed) of a variable extends from its first appearance in a clause through to the end of the clause in an Erlang function. After analyzing the characteristics of an Erlang variable, we have found that any variable in Erlang can be translated to a name in π -calculus with the same spelling and context.

However, like Numbers or Atoms, it has also two kinds of mapping function depending on the context of using.

$$\text{TrPI}_{\text{arg}}(X) := X \quad \text{-(5)}$$

and

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, X) &:= \text{res}'\langle \text{TrPI}_{\text{arg}}(X) \rangle.\underline{\text{nil}} \\ &\quad \text{(5)} \\ &= \text{res}'\langle X \rangle.\underline{\text{nil}} \quad \text{-(6)} \end{aligned}$$

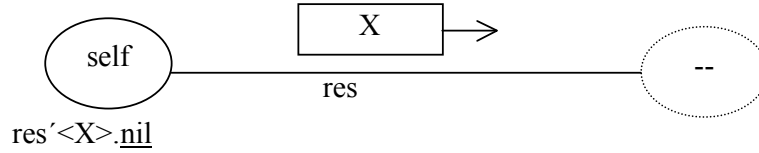


Figure 4.5 Graphical representation of $\text{TrPI}_{\text{exp}}(\text{self}, X)$

Some examples are:

Using rule (5)

$$\text{TrPI}_{\text{arg}}(\text{Start}) := \text{Start}$$

$$\text{TrPI}_{\text{arg}}(\text{End}) := \text{End}$$

$$\text{TrPI}_{\text{arg}}(\text{Start_end}) := \text{Start_end}$$

Using rule (6)

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{Start}) := \text{res}'\langle \text{Start} \rangle.\underline{\text{nil}}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{End}) := \text{res}'\langle \text{End} \rangle.\underline{\text{nil}}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{Start_end}) := \text{res}'\langle \text{Start_end} \rangle.\underline{\text{nil}}$$

Normally variables begin with uppercase letter. Uppercase names are allowed in π -calculus where uppercase and lowercase names are distinct names with same spelling. For example X and x are two distinct names in the π -calculus(cf. Chapter 3).

4.3 Assignment Expression(s)

In PIERlang-00 syntax(Figure 4.1), we have considered two types of assignment expressions, $X=E_1, E_2$ where variable X could be used in expression E_2 and $X=E$, where there is no following expression.

4.3.1 Assignment Expression $X=E_1, E_2$

In Erlang, expressions $X=E_1, E_2$ means that the value of the expression E_1 will be evaluated and then assigned to variable X where X could be further used in the expression E_2 . This could be translated to the π -calculus as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, X = E_1, E_2) := \text{new } \text{expl_res}(\text{TrPI}_{\text{exp}}(\text{self}, E_1) \parallel \text{expl_res}(X).\text{TrPI}_{\text{exp}}(\text{self}, E_2)) \text{ -(7)}$$

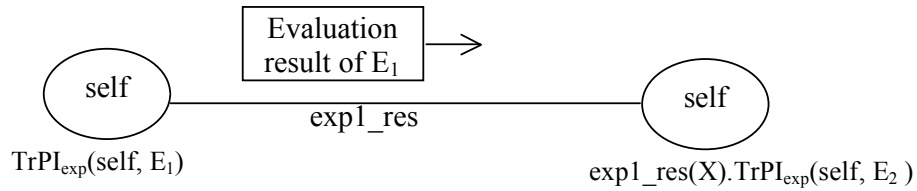


Figure 4.6 Graphical representation of rule(7)

In rule(7), two processes are executed in parallel, one is evaluating the expression E_1 and another one is expecting the value of the expression in channel expl_res . As soon as the result is available in expl_res channel, it will be received by the 2nd process and then execution of remaining expression E_2 will be started. Here we have used the distinguishable channel name expl_res to express clearly that result of evaluation of expression E_1 will be stored in channel expl_res and along that channel expl_res 2nd process will receive the result in X . In this way, evaluation result of E_1 can be used in expression E_2 as well. We have not considered X as a new name as we know that π -calculus, *receive* action (here $\text{expl_res}(X)$) binds X and thus both expl_res and X are bound names for this two parallel processes in π -calculus .

Let us consider an example:

```
Works=do_work,
fun_working(Works).
```

In Erlang, the first expression means that first the variable *Works* will be assigned the atom *do_work*. Then the variable *Works* will be used in the 2nd expression. We have assumed that the definition of the function *fun_working()* has been presented

somewhere else in the program and the above two expressions are executed in a function that has its PID *self*.

Applying rule (7), we can translate these expressions in π -calculus as follows:

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \\
 & \text{Works}=\text{do_work}, \\
 & \text{fun_working}(\text{Works}). \\
 & (7) \\
 & =\text{new exp1_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{Works}=\text{do_work}) \parallel \text{exp1_res}(\text{Works}).\text{TrPI}_{\text{exp}}(\text{self}, \\
 & \quad \text{fun_working}(\text{Works}))) \\
 & =(\text{cf. later sections})
 \end{aligned}$$

4.3.2 Assignment Expression $X=E$

Another simple assignment expression is denoted as $X=E$ where X is a variable and E is an Expression. The evaluation result of E will be assigned to X after executing the expression. This X then may or may not be used in the subsequent expression(s). We have written the corresponding translation mapping function for expression $X=E$ by sending the evaluation result of expression E along the global channel *res*. Thereby, any process having *receive* action along channel *res* can receive this result. As there is no expression(s) following E , we have considered it in the following way:

$$\text{TrPI}_{\text{exp}}(\text{self}, X = E) := \text{TrPI}_{\text{exp}}(\text{self}, E) \parallel \text{res}(X).\underline{\text{nil}} \quad -(8)$$

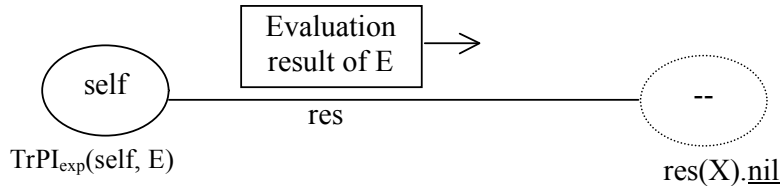


Figure 4.7 Graphical representation of rule(8)

We have also included the global name *self* in the input parameters list along with the expression in order to provide the PID of the process that is currently executing the expression.

In the translation mappings, the expression E is translated first, result of evaluation of E will be ultimately sent along the channel *res* depending on the context. After that a parallel process has been included that will be used to receive data along *res* and binds

the received data with name X . In this way, the value of the expression is bound with the name X in the π -calculus.

Let us consider an example,

$Y = \text{dowork}$

In PIERlang, this expression means that after executing it, the atom *dowork* will be assigned to the variable Y where *self* is the PID of the process that executes the expression.

Using rule (8), we can translate this expression into π -calculus as follows (here Y corresponds to X and *dowork* corresponds to E in rule (8)):

$$\begin{aligned} & (8) \\ \text{TrPI}_{\text{exp}}(\text{self}, Y=\text{dowork}) &= \text{res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{dowork}) \parallel \text{res}(Y) . \underline{\text{nil}}) \\ (4) \\ &= \text{res}(\text{res}'\langle \text{dowork} \rangle . \underline{\text{nil}} \parallel \text{res}(Y) . \underline{\text{nil}}) \end{aligned}$$

The result of evaluating *dowork* is sent along channel *res* and a parallel process is trying to receive the data along the same channel *res* by which the value of the expression *dowork* will be bound with the name Y .

4.4 Send Expression: PID as Implicit Mailbox

In PIERlang-00 syntax(Figure 4.1), only variables, atoms and numbers are used as the arguments of the *send* expression. The send expression is $A_1! A_2$ where A_1 and A_2 are place holder for Arguments(number, atoms and variables).

For a *send* expression $A_1! A_2$, in Erlang, we know that A_1 is a PID and A_2 is a message that will be sent to the mailbox of the process identified by the PID stored in A_1 and the value of the expression $A_1! A_2$ is the value of the A_2 .

The corresponding π -calculus translation would be as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, A_1! A_2) := (\text{TrPI}_{\text{arg}}(A_1))' < \text{TrPI}_{\text{arg}}(A_2) > . \underline{\text{nil}} \parallel \text{res}' < \text{TrPI}_{\text{arg}}(A_2) > . \underline{\text{nil}} \quad -(9)$$

Translation of *send* expression has introduced two processes working in parallel; one is a direct mapping of the *send* expression in the π -calculus and another is sending the message A_2 to the *res* channel so that any other process waiting for a message from *res* channel can receive that A_2 with a *receive* expression along *res*. In this way, the message is sent to the specific process identified by PID A_1 and along the *res* channel.

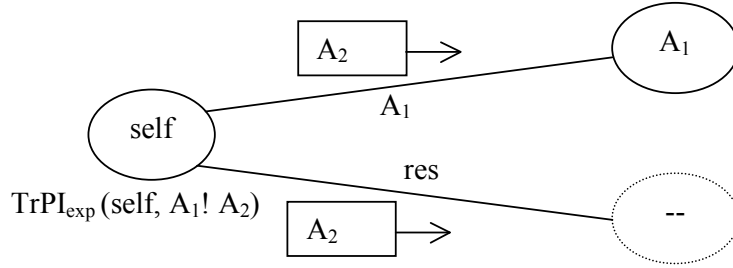


Figure 4.8 Graphical representation of rule(9)

Of course, it is needed to call the TrPI_{arg} function to have a corresponding π -calculus translation of arguments of the send expression.

Lets consider an example that has been used in previous section:

$\text{DestPid} ! \text{dowork};$

In Erlang, this *send* expression means that atom *dowork* would be sent to the mailbox of the process identified by the PID stored in the variable *DestPid*.

Using rule (9), we can obtain a corresponding π -calculus representation of the above expression as follows:

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \text{DestPid} ! \text{dowork}) \stackrel{(9)}{:=} (\text{TrPI}_{\text{arg}}(\text{DestPid}))' < \text{TrPI}_{\text{arg}}(\text{dowork}) > . \underline{\text{nil}} \\
 & \quad \parallel \text{res}' < \text{TrPI}_{\text{arg}}(\text{dowork}) > . \underline{\text{nil}} \\
 & \stackrel{(5) \ \& \ (3)}{=} \text{DestPid}' < \text{dowork} > . \underline{\text{nil}} \parallel \text{res}' < \text{dowork} > . \underline{\text{nil}}
 \end{aligned}$$

Here in the π -calculus representation, we have seen that the atom *dowork* is sent along channel *DestPid* and along to the channel *res*.

In PIErlang , this *DestPid* is the PID of the process to which the atom *dowork* is sent as message. Normally in PIErlang , message(s) is stored first in the mailbox of the destination process(here *DestPid*). In π -calculus, we have done this by presenting *DestPid* (PID of the destination process to which message has to be sent) as a channel and sending the message over this channel thus, implicitly representing the mailbox of the receiving process where the channel name *DestPid* is used as an implicit mailbox of the process whose PID is *DestPid*. In this way, while receiving the message, the receiver process (here same *DestPid*) will receive the message along channel name *DestPid*. The message is also sent over the *res* channel so that any process having the

same *res* channel can receive the message. We will discuss about receiving message in details in the following sections 4.7 and 4.9.

Similarly,

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \text{RemotePid} ! 6) \stackrel{(9)}{:=} (\text{TrPI}_{\text{arg}}(\text{RemotePid}))' < \text{TrPI}_{\text{arg}}(6) > .\underline{\text{nil}} \parallel \\
 & \quad \text{res}' < \text{TrPI}_{\text{arg}}(6) > .\underline{\text{nil}} \\
 & (5) \ \& \ (1) \\
 & = \text{RemotePid}' < \text{unknown} > .\underline{\text{nil}} \parallel \text{res}' < \text{unknown} > .\underline{\text{nil}}
 \end{aligned}$$

In this example, one interesting thing is the introduction of unknown name in π -calculus representation. Here, we see that a number 6 is sent to the process identified by the variable *RemotePid*. But as we know, we do not have any semantic representation of any number in π -calculus, we have called the rule (1) to use an *unknown* name instead of the number 6.

4.5 Function Call Expressions

In the expression of our PIERlang-00 syntax, we have introduced 2 different kinds of function calls: n-ary function and n-ary spawn function.

4.5.1 n-ary Function Call

n-ary function call is an expression in Erlang. We have translated an n-ary Erlang function call to (n+1)-ary process call in the π -calculus where the first argument will be *self*, a new name representing the PID of the process executing the function expression. Translation of n-ary function is as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, f(A_1, A_2, \dots, A_n)) := f(\text{self}, \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n)) \quad -(10)$$

Only Arguments i.e. variables, numbers and atoms can be used as the arguments of the function expression. To have a proper translation depending on the types of the arguments, the TrPI_{arg} translation function is applied for each of the arguments.

Similarly, for the 0-ary function expression, we can have the following translation mapping:

$$\text{TrPI}_{\text{exp}}(\text{self}, f()) := f(\text{self}) \quad -(10A)$$

Let us consider a simple example:

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \text{start}(\text{Sunday}, \text{work}, 5)) \stackrel{(10)}{:=} \text{start}(\text{self}, \text{TrPI}_{\text{arg}}(\text{Sunday}), \text{TrPI}_{\text{arg}}(\text{work}), \\
 & \quad \text{TrPI}_{\text{arg}}(5))
 \end{aligned}$$

$$\begin{aligned}
 & (5) \ (3) \ \& \ (1) \\
 & = \text{start}(\text{self}, \text{Sunday}, \text{work}, \text{unknown})
 \end{aligned}$$

Similarly,

$$\begin{aligned} & (10A) \\ \text{TrPI}_{\text{exp}}(\text{self}, \text{start}()) &:= \text{start}(\text{self}) \end{aligned}$$

4.5.2 n-ary Spawn Call

Spawn call is like a function call, but instead of having the current process executes the call, a new process is constructed. The child process will run in parallel with the current, performing the function call. *Spawn* returns the PID of the newly created process that executes the function which is used as a parameter of *spawn* call. If the function call terminates, the process executing it will end. If the function call returns a value, this value will be ignored. In PIERlang, *spawn* is simplified omitting the module in which the function is placed.

The syntax of the restricted form *spawn* is as follows:

$$\text{spawn}(f, [A_1, A_2, \dots, A_n]) \quad ; n \geq 0$$

Like function call arguments, only numbers, atoms and variables are used as the arguments of the function of the *spawn* call.

Again, depending on the context we have two versions of *spawn*; one is storing the newly returned PID in a variable and another one is ignoring it. First, we consider the *spawn* where the newly constructed PID is stored in a variable for further use in the subsequent expression(s). A translation mapping of such a *spawn* call is provided below:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, X = \text{spawn}(f, [A_1, \dots, A_n]), E) &:= \text{new fpid, p, f_res}(p' < \text{fpid} > . \underline{\text{nil}} \parallel \\ \text{TrPI}_{\text{exp}}(\text{fpid}, f(A_1, \dots, A_n)) &\parallel f_res(\text{dummy}). \underline{\text{nil}} \parallel p(X). \text{TrPI}_{\text{exp}}(\text{self}, E) \end{aligned} \quad -(11)$$

As before, *self* is included along with the *spawn* as one of the arguments of the translation function to represent that the *spawn* function is executed by the current process i.e. the process which has PID *self()*. After executing the *spawn*, the newly created PID will be stored in the variable *X* and the function *f* will be executed by newly created process in parallel with the main process(PID *self()*) where *spawn* function is embedded.

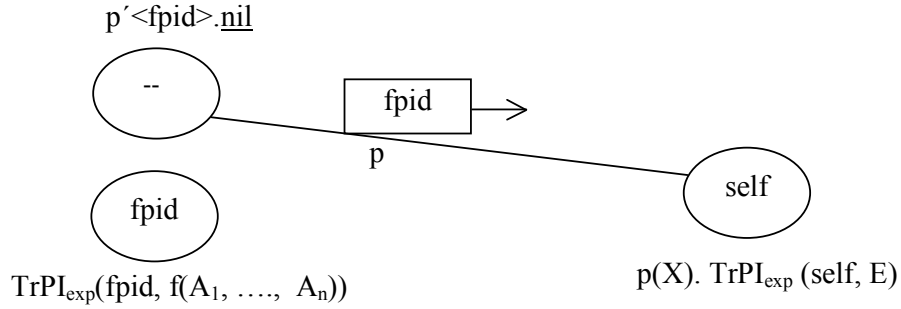


Figure 4.9 Graphical representation of rule(11)

In the translation mappings, three fresh names $fpid$, p and f_res have been created where $fpid$ is supposed to be the PID of newly created process which will execute function f , p is used to *send/receive* the $fpid$ and f_res is the channel through the result of the evaluation of *spawn* will be sent. Bound name X is used to receive the $fpid$ along p for further use.

In rule(11), one process sends($p' \langle fpid \rangle . \underline{nil}$) the PID $fpid$ along p and the same PID is used in the translation mapping of the function $f(A_1, \dots, A_n)$ by the second parallel process($TrPI_{exp}(fpid, f(A_1, \dots, A_n))$) which clearly specifies that newly created process with PID $fpid$ will execute the function f . There are also two more processes working in parallel; one of which waits to receive something(return value of function call) from the f_res channel and then discards the received value and another one waits to receive $fpid$ in name X . After receiving $fpid$ in X , the current process (here *self*) will execute the remaining expression E . In this way, along with reaction rule of π -calculus, if there is any X in expression E that will be replaced by the PID $fpid$ which completely meets the context of the *spawn* function and our intention of translation mapping. It could be explained more clearly with an example as follows:

```
SenderID=spawn(fun_working, [Day_start, tea_break, 5]),
SenderID ! dowork;
```

This is a part of a PIERlang program where *spawn* function is called with the function *fun_working* having 3 arguments. The definition of the function *fun_working* has been omitted here with a view to have simplicity in translation. The new PID created by the *spawn* function is assigned to variable *SenderID* and to that process atom *dowork* will be sent. Using rule (11), we can have a corresponding π -calculus translation as follows:

```
TrPI_exp(self, SenderID=spawn(fun_working, [Day_start, tea_break, 5]),
SenderID ! dowork; )
```

(11)

$\text{:= new fpid, p, fun_working_res}(p' < \text{fpid} > .\underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{fpid},$
 $\text{fun_working}(\text{Day_start}, \text{tea_break}, 5)) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel$
 $p(\text{SenderID}).\text{TrPI}_{\text{exp}}(\text{self}, \text{SenderID} ! \text{dowork}))$

(10)

$\text{= new fpid, p, fun_working_res}(p' < \text{fpid} > .\underline{\text{nil}} \parallel \text{fun_working}(\text{fpid},$
 $\text{TrPI}_{\text{arg}}(\text{Day_start}), \text{TrPI}_{\text{arg}}(\text{tea_break}), \text{TrPI}_{\text{arg}}(5)) \parallel \text{fun_working_res}$
 $(\text{dummy}).\underline{\text{nil}} \parallel p(\text{SenderID}).\text{TrPI}_{\text{exp}}(\text{self}, \text{SenderID} ! \text{dowork}))$

(5) (3) & (1)

$\text{= new fpid, p, fun_working_res}(p' < \text{fpid} > .\underline{\text{nil}} \parallel \text{fun_working}(\text{fpid}, \text{Day_start},$
 $\text{tea_break}, \text{unknown})) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel p(\text{SenderID}).$
 $\text{TrPI}_{\text{exp}}(\text{self}, \text{SenderID} ! \text{dowork}))$

(9)

$\text{= new fpid, p, fun_working_res}(p' < \text{fpid} > .\underline{\text{nil}} \parallel \text{fun_working}(\text{fpid},$
 $\text{Day_start}, \text{tea_break}, \text{unknown})) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel$
 $p(\text{SenderID}).(\text{TrPI}_{\text{arg}}(\text{SenderID}))' < \text{TrPI}_{\text{arg}}(\text{dowork}) > .\underline{\text{nil}} \parallel \text{res}' <$
 $\text{TrPI}_{\text{arg}}(\text{dowork}) > .\underline{\text{nil}})$

(5) & (3)

$\text{= new fpid, p, fun_working_res}(p' < \text{fpid} > .\underline{\text{nil}} \parallel \text{fun_working}(\text{fpid},$
 $\text{Day_start}, \text{tea_break}, \text{unknown})) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel$
 $p(\text{SenderID}).(\text{SenderID}' < \text{dowork} > .\underline{\text{nil}} \parallel \text{res}' < \text{dowork} > .\underline{\text{nil}}))$

If we now apply the reaction rule of π -calculus on channel p then it will look like as follows:

(react on p)

$\text{=> new fpid, p, fun_working_res}(\underline{\text{nil}} \parallel \text{fun_working}(\text{fpid}, \text{Day_start}, \text{tea_break},$
 $\text{unknown})) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel (\text{fpid}' < \text{dowork} > .\underline{\text{nil}} \parallel$
 $\text{res}' < \text{dowork} > .\underline{\text{nil}}))$

By doing so, name *SenderID* is bound to PID *fpid* (outcome of *spawn* that executes function *fun_working*) which could be used in the remaining expression.

In the 2nd type of *spawn*, the newly created PID is ignored rather storing it to a variable. It has been assumed that the PID of the newly created process will no longer be used in the subsequent expression(s). In the following, there is such kind of translation mapping:

$\text{TrPI}_{\text{exp}}(\text{self}, \text{spawn}(f, [A_1, \dots, A_n]), E) \text{:= new fpid, f_res}(\text{TrPI}_{\text{exp}}(\text{fpid}, f(A_1, \dots,$
 $A_n)) \parallel \text{f_res}(\text{dummy}).\underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{self}, E))$ -(12)



Figure 4.10 Graphical representation of rule(12)

The only major difference between this rule (12) and the rule (11) is that in rule (12), the remaining expression E is executed without waiting to receive the new PID $fpid$ along p since it has been assumed that evaluation of E will no longer be needed of the PID $fpid$.

Lets consider a part of PIERlang program as follows:

```
spawn(fun_working, [Day_start, tea_break, 5]),
Friday = half_office;
```

With this partial program code, it has been noticed that PID created by *spawn* is not stored to any variable as it will not be used in the subsequent expression(s) E (Here $\text{Friday} = \text{half_office}$).

Using rule (12), we can have the corresponding π -calculus representation as follows:

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \text{spawn}(\text{fun_working}, [\text{Day_start}, \text{tea_break}, 5]), \\
 & \quad \text{Friday} = \text{half_office};) \\
 & \quad (12) \\
 & := \underline{\text{new}} \text{ fpid}, \text{fun_working_res}(\text{TrPI}_{\text{exp}}(\text{fpid}, \text{fun_working}(\text{Day_start}, \text{tea_break}, 5)) \parallel \\
 & \quad \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{self}, \text{Friday} = \text{half_office})) \\
 & \quad (10) \\
 & = \underline{\text{new}} \text{ fpid}, \text{fun_working_res}(\text{fun_working}(\text{fpid}, \text{TrPI}_{\text{arg}}(\text{Day_start}), \\
 & \quad \text{TrPI}_{\text{arg}}(\text{tea_break}), \text{TrPI}_{\text{arg}}(5)) \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{self}, \\
 & \quad \text{Friday} = \text{half_office})) \\
 & \quad (5) (3) \& (1) \\
 & = \underline{\text{new}} \text{ fpid}, \text{fun_working_res}(\text{fun_working}(\text{fpid}, \text{Day_start}, \text{tea_break}, \text{unknown}) \\
 & \quad \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{self}, \text{Friday} = \text{half_office})) \\
 & \quad (7) \\
 & = \underline{\text{new}} \text{ fpid}, \text{fun_working_res}(\text{fun_working}(\text{fpid}, \text{Day_start}, \text{tea_break}, \text{unknown}) \\
 & \quad \parallel \text{fun_working_res}(\text{dummy}).\underline{\text{nil}} \parallel (\underline{\text{new}} \text{ assn_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{half_office}) \\
 & \quad \parallel \text{assn_res}(\text{Friday}).\underline{\text{nil}}))) \\
 & \quad (3)
 \end{aligned}$$

```
= new fpid, fun_working_res, assn_res (fun_working(fpid, Day_start,
    tea_break unknown) || fun_working_res(dummy).nil ||
    ((assn_res'<half_office>.nil || assn_res(Friday).nil))
```

4.6 Sequence of Expressions

In PIErlang, it has been presented that an expression could be also a composition of two independent sub-expressions. This is formally denoted as E_1, E_2 where evaluation of E_2 is independent of evaluation of E_1 but E_2 will be evaluated only after evaluation of E_1 . The corresponding π -calculus mapping of such sequence of expressions is formed as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, E_1, E_2) := \text{new } \text{expl_res} (\text{TrPI}_{\text{exp}}(\text{self}, E_1) || (\text{expl_res}(\text{dummy}). \text{TrPI}_{\text{exp}}(\text{self}, E_2))) \quad \text{-(13)}$$

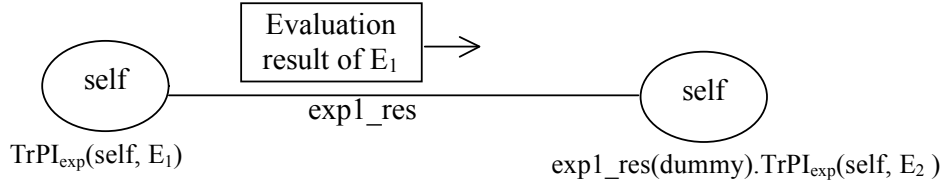


Figure 4.11 Graphical representation of rule(13)

From the π -calculus representation in rule(13), it is clear that expression E_1 will be evaluated first. In the 2nd process, we see that there is a receive action along *expl_res* before evaluating expression E_2 . This receive action can not be performed without evaluation of E_1 , as E_1 is the only candidate that can send its result along channel *expl_res*. As soon as E_1 is evaluated and its result is passed through channel *expl_res*, receive action *expl_res(dummy)* will be performed with the reaction rule of π -calculus and then evaluation of E_2 will be started, thus forcing E_2 to be evaluated after E_1 . Here it has to be noticed that we have discarded the received value (in dummy) as it is not needed in evaluation of E_2 .

Lets consider the following two expressions which are independent of each other and we want to keep the order of evaluation as it is in the context.

```
do_work,
go_market ;
```

The corresponding π -calculus representation using rule (13) will be as follows:

TrPI_{exp}(self, do_work,
go_market;)

(13)

:= new exp1_res(TrPI_{exp}(self, do_work) ||
exp1_res(dummy).TrPI_{exp}(self, go_market))

(4)

= new exp1_res(exp1_res'<do_work>.nil ||
exp1_res(dummy).TrPI_{exp}(self, go_market))

(4)

= new exp1_res, exp2_res (exp1_res'<do_work>.nil ||
exp1_res(dummy).exp2_res'<go_market>.nil)

Here it has been seen that although the 2nd process is discarding the received result of evaluation of expression E₁, the evaluation of expression E₂ is now in order as we expected i.e. evaluating E₂ after E₁.

4.7 Receive Expression

In Section 4.4, we have discussed the message (only a number or an atom or a variable) passing mechanism to a specific process, where we have considered the process identifier of the process to which the message is to sent as the channel along which the potential message has to be sent. Consider the example given in Section 4.4,

DestPid ! dowork;

We got the translation mapping in π -calculus as follows,

DestPid'< dowork>.nil || res'< dowork>.nil

The message *dowork* is sent over the channel *DestPid* and also along the *res* channel. We have also discussed (cf. Section 4.4) that this representation is an implicit representation of the process's (here *DestPid*) mailbox.

Now consider that receiver process (here *DestPid* is the process identifier of the receiver process) is ready to receive message along channel *DestPid* and after receiving any message along *DestPid*, it would be in state Q. This receiving scenario can be written in π -calculus as follows:

DestPid(input_pat).Q

The bound name *input_pat* will be bound to Q with the corresponding receiving element along channel *DestPid*.

Now consider that both the sender and receiver processes are working in parallel to have a communication between them.

DestPid'<dowork>.nil || res'<dowork>.nil || **DestPid**(input_pat).Q

Now using the reaction rule of π -calculus we can obtain,

(react on **DestPid**)
 \Rightarrow nil || res'<dowork>.nil || Q[input_pat/dowork]

The receiver process will now work as Q by replacing every free occurrence of *input_pat* in Q by *dowork*.

There is still another process which sends the message over the global result channel *res* with a view to enable any other process to receive that sent message along the same *res* channel. However, use of *res* is completely dependent on the context of using. We will present more about this with some examples later in Chapter 5.

Until now, we have discussed so far the sending and basic receiving mechanism of PIERlang in π -calculus perspective. The sending of message discussed in Section 4.4 is sound enough to meet the presented send expression. However, the receiving mechanism in PIERlang is not so straight forward. The general syntax that we have presented in PIERlang is resembled as follows:

```
receive
  Pattern1 -> Body1;
  ...;
  PatternN -> BodyN
end
```

The receiver process receives messages sent to its mailbox by any process. The patterns *Pattern* are sequentially matched against the first message in time order in the mailbox, then the second, and so on. If a match succeeds the corresponding *Body* is evaluated. The matching message is consumed, that is removed from the mailbox, while any other messages in the mailbox remain unchanged. The return value of *Body* is the return value of the *receive* expression. *Receive* never fails. Execution is suspended, possibly indefinitely, until a message arrives that does match one of the patterns.

4.7.1 Matches of Receive Expression

Receive has the branching feature of a *case* expression and also has to match the reception semantics of PIERlang. Because of the quite inadequate statements available

in π -calculus, the translation is not so accurate to meet reception semantics. We have chosen the asynchronous π -calculus as our target specification language where the order of the received messages could not be respected. It was beyond the scope of this thesis to model the mailbox explicitly to have the full semantics of a *receive* statement using synchronous π -calculus. We have considered that the non-deterministic choices among the translated matches of the *receive* statement would be the translation of a *receive* statement in the π -calculus.

$$\begin{aligned} &\text{TrPI}_{\text{exp}}(\text{self}, \text{receive } M_1; \dots; M_n \text{ end}) \\ &:= \text{TrPI}_{\text{match}}(\text{self}, M_1) + \dots + \text{TrPI}_{\text{match}}(\text{self}, M_n) \end{aligned} \quad -(14)$$

The whole *receive* expression is splitted into the non-deterministic choices of the single match expressions. We also see that if the name *self* is used for the whole *receive* expression, then *self* is also used in the *single-match* expressions too. As before, *self* is the process identifier of the process that executes the *receive* expression.

Let us consider an example,

```
receive
  Monday -> work;
  saturday -> take_rest
end
```

We can present this receive expression using rule (14) as follows:

$$\begin{aligned} &\text{TrPI}_{\text{exp}}(\text{self}, \text{receive} \\ &\quad \text{saturday -> take_rest} \\ &\quad \text{Monday -> work;} \\ &\quad \text{end}) \\ (14) \\ &= \text{TrPI}_{\text{match}}(\text{self}, \text{Monday -> work}) + \text{TrPI}_{\text{match}}(\text{self}, \text{saturday -> take_rest}) \end{aligned}$$

From this simple example, it is now clear that we have decomposed (as nondeterminism) the *receive* expression into two smaller *Match* expressions, each expression is for each *match* of the *receive* expression. Another function named $\text{TrPI}_{\text{match}}$ is used for translation mapping each of the *single-match* expressions. We will now try to present the $\text{TrPI}_{\text{match}}$ function formally with several small examples.

4.7.2 Match

In PI Erlang, a match is of the form $P \rightarrow E$, where pattern P could be a variable, or an atom or a number only. We have introduced pattern matching in *receive* and *case*

expressions. In this section, we will discuss about the pattern matching translations for the *receive* expression only.

We have tried to translate such *match* in the π -calculus with $\text{TrPI}_{\text{match}}$ function which has the following signature.

$\text{TrPI}_{\text{match}} : \text{Name X Match} \rightarrow \text{Process}$

In the definition of $\text{TrPI}_{\text{match}}$, a *name* along with the *match* are used as the Input and as Output, a process in the π -calculus is produced. Here *name* is the so called *self*, the process identifier of the process executing the *receive* expression.

4.7.2(a) Match: Atoms as Patterns

We have applied the *name matching* feature of the π -calculus to handle pattern matching of PIERlang. First, consider the match of the form $a \rightarrow E$. Here the pattern a is an atom and expression E could be any valid expression built from PIERlang-00.

Here is the translation mapping of such a pattern:

$\text{TrPI}_{\text{match}}(\text{self}, a \rightarrow E)$

$:= \text{self}(\text{input_pat}).[\text{input_pat} = \text{TrPI}_{\text{arg}}(a)] \text{TrPI}_{\text{exp}}(\text{self}, E)$

Match $a \rightarrow E$ with the so-called name *self* are provided to the $\text{TrPI}_{\text{match}}$ function as Input. As Output, along *self* target message will be received in *input_pat* which will be matched against the pattern (here with atom a) of the given match. As we have only considered atoms, numbers or variables to be a pattern in the match, we also need to call the TrPI_{arg} function to the pattern. Using rule (3) of arguments translation, the final translation mapping for atom as pattern can be written as follows:

$\text{TrPI}_{\text{match}}(\text{self}, a \rightarrow E)$

$:= \text{self}(\text{input_pat}).[\text{input_pat} = a] \text{TrPI}_{\text{exp}}(\text{self}, E)$ -(15)

Receiving action $\text{self}(\text{input_pat})$ binds the name *input_pat*, it is not required to declare *input_pat* as a new name during translation mapping.

Let us consider a simple example:

```
receive
  monday -> Work = 8
end
```

This program segment indicates that if the potential input message is an atom named *monday* then it will be matched with the given pattern *monday*, consequently the

expression $Work=8$ will be evaluated. Applying rule(15), we can have a π -calculus representation of this program segment as follows:

```
TrPIexp(self, receive
```

```
    monday -> Work = 8
```

```
end )
```

(14)

```
=TrPImatch(self, monday -> Work = 8 )
```

(15)

```
=self(input_pat).[input_pat=monday]TrPIexp(self, Work = 8)
```

(8)

```
= new exp_res (self(input_pat).[input_pat=monday](TrPIexp(self, 8) ||  
    exp_res(Work).nil))
```

(2)

```
:= new exp_res (self(input_pat).[input_pat=monday](exp_res'<unknown>.nil ||  
    exp_res(Work).nil))
```

4.7.2(b) Match: Numbers as Patterns

Like atoms in Section 4.7.2(a), for matching of numbers we get the following translation mapping function:

```
TrPImatch(self, n -> E )
```

```
:= self(input_pat).[input_pat = TrPIarg(n)] TrPIexp(self, E)
```

Applying rule (1), we get the following translation mapping function for numbers as patterns:

```
TrPImatch (self, n -> E )
```

```
:=self(input_pat).[input_pat = unknown] TrPIexp(self, E) -(16)
```

Let us consider another program segment,

```
receive
```

```
    10 -> Work =do
```

```
end
```

From the code segment, it is clear that if the input pattern is 10, then atom *do* will be assigned to variable *Work*.

Applying rule (16), the corresponding π -calculus representation would be as follows:

```
TrPIexp(self, receive
          10 -> Work =do
          end)
```

(14)
=TrPI_{match}(self, 10 -> Work =do)

(16)
:=self(input_pat).[input_pat=unknown] TrPI_{exp}(self, Work =do)

(8)
=self(input_pat).[input_pat=unknown] (TrPI_{exp}(self, do) || res(Work).nil)

(4)
=self(input_pat).[input_pat=unknown] (res'<do>.nil || res(Work).nil)

According to rules (15) and (16), a message(a number or an atom) will be sent by the current process or by some other processes along the *self* channel and that message will be received here, matched against the given pattern, if name matching is successful then the remaining body expression E will be evaluated.

4.7.2(c) Match: Variables as Patterns

With rules (15) and (16) we can only deal with atoms and numbers as patterns. As already mentioned, variable can match with any term, therefore, no name matching is required for a variable as pattern in our single-match translation mapping. However, this variable pattern could be used in the body expression E where the corresponding received message is expected to be substituted in place of the variable. Thereby, we have used the substitution feature of the π -calculus on the corresponding bound name with the variable in Body expression E. The translation mapping for variable as pattern has been presented as follows:

```
TrPImatch (self, X -> E)
:= self(input_pat).(TrPIexp(self, E)[X/input_pat] )
```

-(17)

Where [X/input_pat] denotes the replacement of every free occurrences of X in translated E by *input_pat*.

Potential message(sent term) will be received along the *self* channel in *input_pat* and evaluation of E will be started. Name matching is not required here, since variable pattern can be matched with any received term. The variable pattern X could be used

in expression E , thereby, the substitution feature of π -calculus is applied to replace X with the received message. However, we can consider a more simplified version as follows:

$$\begin{aligned} & \text{TrPI}_{\text{match}}(\text{self}, X \rightarrow E) \\ & := \text{self}(X).\text{TrPI}_{\text{exp}}(\text{self}, E) \end{aligned} \quad \text{-(17A)}$$

We notice that we have used the same variable name X as the received name along channel *self*. The intention of doing so is that variable X could be used in the evaluation of E and therefore, we have bound the received term in X so that while evaluating E , received term will be used in place of X . We have not used name matching here as well.

Consider an example of variable pattern as follows:

```
receive
  Monday -> Duties = Monday
end
```

This program segment indicates that whatever the input pattern is, expression *Duties = Monday* will be evaluated as variable pattern can be matched with any term.

The corresponding π -calculus representation can be obtained using rule (17A) as follows:

```
TrPIexp(self,
  receive
    Monday -> Duties = Monday
  end )
```

(14)
 $= \text{TrPI}_{\text{match}}(\text{self}, \text{Monday} \rightarrow \text{Duties} = \text{Monday})$

(17A)
 $= \text{self}(\text{Monday}).\text{TrPI}_{\text{exp}}(\text{self}, \text{Duties} = \text{Monday})$

(8)
 $= \text{self}(\text{Monday}).(\text{TrPI}_{\text{exp}}(\text{self}, \text{Monday}) \parallel \text{res}(\text{Duties}).\underline{\text{nil}})$

(6)
 $= \text{self}(\text{Monday}).(\text{res}'\langle \text{Monday} \rangle.\underline{\text{nil}} \parallel \text{res}(\text{Duties}).\underline{\text{nil}})$

Now we see that along *self* input term will be received and bound in variable *Monday*. After that with the reaction rule of π -calculus, content of *Monday* will be bound with

variable *Duties* which was the same intention as we expected from PIERlang code segment. We will discuss in details about variable pattern in Chapter 5.

Finally, consider another example with two matches:

```
TrPIexp(self, receive
    saturday -> take_rest ;
    Monday -> work
end )
```

(14)

$$:= \text{TrPI}_{\text{match}}(\text{self}, \text{saturday} \rightarrow \text{take_rest}) + \text{TrPI}_{\text{match}}(\text{self}, \text{Monday} \rightarrow \text{work})$$

(15) (17A)

$$= \text{self}(\text{input_pat}).[\text{input_pat} = \text{saturday}] \text{TrPI}_{\text{exp}}(\text{self}, \text{take_rest}) +$$

$$\text{self}(\text{Monday}).\text{TrPI}_{\text{exp}}(\text{self}, \text{work})$$

(4)

$$= \text{self}(\text{input_pat}).[\text{input_pat} = \text{saturday}] \text{res}' \langle \text{take_rest} \rangle . \underline{\text{nil}} +$$

$$\text{self}(\text{Monday}).\text{res}' \langle \text{work} \rangle . \underline{\text{nil}}$$

4.8 Case Expression

The general form of a *case* expression in PIERlang is as follows:

```
case E of
    P1 -> E1;
    .... ;
    Pn -> En
end
```

The *case head* expression E is evaluated and the patterns P₁..P_n are sequentially matched against the result. If a match succeeds the corresponding body expression is evaluated. For instance, if the evaluation result of *case head* E is matched with pattern P₃, corresponding body expression E₃ will be evaluated. The return value of *body* expression is the return value of the *case* expression. If there is no matching pattern, a *case_clause* run-time error will be occurred. To avoid such run-time error, we consider that in *case* expression, there will be at least one matching pattern.

4.8.1 Case: As a Sequence of Two Expressions

To have an easier translation mapping rule for *case* expression, it is rewritten as the sequence of two sub-expressions as follows:

```

X=E,
case X of
  P1->E1;
  .... ;
  Pn->En
end

```

Case expression is rewritten as a sequence of two expressions where the first one is an assignment expression of returning the evaluation result of *case head* expression E to an unbound variable X and the second expression is again a *case* expression where in *case head*, bound variable X (as it has been bound in previous assignment expression) is used. While working with a bound variable in *case head*, translation mapping seems to be very straight forward as we can directly use name matching feature for X against the patterns of the matches of the *case* expression. However, in this case, the evaluation result of *case head* expression E must be a single value like a number, an atom or a bound variable. We can formally present the mapping rule as follows:

```

TrPIexp(self, case E of
  P1->E1;
  .... ;
  Pn->En
end)

```

```

:=TrPIexp(self, X=E,
  case X of
    P1->E1;
    .... ;
    Pn->En
  end )

```

```

(7)
:=new expl_res(TrPIexp(self, E) || expl_res(X).TrPIexp(self,
  case X of
    P1->E1
    .... ;
    Pn->En
  end ) )

```

```

:=new expl_res( TrPIexp(self, E) || expl_res(X).( [X=TrPIarg(P1)] TrPIexp(self, E1)
  + ..+ [X=TrPIarg(Pn)] TrPIexp(self, En))
-(18)

```

It has to be noted that evaluation result of E is to be sent along *exp1_res* as per rule the rule of sequence of expressions(rule(7)). When a variable is as pattern in any of the matches, the corresponding name matching expression will be ignored in rule (18). This pattern variable could be used in the body expression which is not handled in rule (18).

Consider another example where *case head* is a 3-ary function call,

```
case fun_working(Start, End, Busy) of
  true -> dowork;
  false -> take_rest;
  undefined -> sleep
end.
```

Using rule (18), its translation will be as follows:

```
TrPIexp(self, case fun_working(Start, End, Busy) of
  true -> dowork;
  false -> take_rest;
  undefined -> sleep
end.)
```

```
:=TrPIexp(self, X=fun_working(Start, End, Busy),
  case X of
    true -> dowork;
    false -> take_rest;
    undefined -> sleep
  end.)
```

(18) & (3)

```
= new exp1_res(TrPIexp(self, fun_working(Start, End, Busy) ) || exp1_res(X).(
  [X=true] TrPIexp(self, dowork) + [X=false] TrPIexp(self, take_rest) +
  [X=undefined] TrPIexp(self, sleep) ) )
```

(10) & (5)

```
= new exp1_res(fun_working(self, Start, End, Busy) || exp1_res(X).( [X=true]
  TrPIexp(self, dowork) + [X=false] TrPIexp(self, take_rest) + [X=undefined]
  TrPIexp(self, sleep) ) )
```


(4)

$$= \text{new } \text{expl_res}(\text{fun_working}(\text{self}, \text{Start}, \text{End}, \text{Busy}) \parallel \text{expl_res}(X).([X=\text{true}] \\ \text{res}'\langle\text{dowork}\rangle.\underline{\text{nil}} + [X=\text{false}] \text{res}'\langle\text{take_rest}\rangle.\underline{\text{nil}} + [X=\text{undefined}] \\ \text{res}'\langle\text{sleep}\rangle.\underline{\text{nil}}))$$

While evaluating process $\text{fun_working}(\text{self}, \text{Start}, \text{End}, \text{Busy})$, its evaluation result will not be sent along the fun_working_res channel as per the global invariant in Section 4.1.4, rather, will be sent along expl_res channel. The 2nd process is waiting to receive something along channel expl_res in X , therefore, as soon as the evaluation result of the function fun_working is sent along expl_res , with reaction rule of π -calculus that result will be bound with name X and then X will be used for name matching and if matching is successful with any of the patterns corresponding *body* expression will be returned as the result of the *case* expression in *res* channel.

4.8.2 Case: As a Modified Receive Expression

Rule (18) works fine when the patterns of the Matches are numbers or atoms. It cannot handle variable pattern accurately, especially, when the pattern variable is used in the corresponding *body* expression. The syntax of *case* expression resembles *receive* expression in the sense of using Matches. Matches in the *case* expression can be tackled with the same way as presented for *receive* expression in Sections 4.7.2(a), 4.7.2(b) and 4.7.2(c). The issue of using variable as pattern in Matches is presented in Section 4.7.2(c) for *receive* expression Matches. Therefore, if we can map the *case* expression to the *receive* expression, translation mapping will be easier and accurate. In the following, there is a formal representation of such mapping:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{case } E \text{ of } M_1; \dots; M_n \text{ end}) &:= \text{new } \text{case_res} (\text{TrPI}_{\text{exp}}(\text{self}, E) \parallel \\ &\quad \text{TrPI}_{\text{exp}}(\text{case_res}, \text{receive } M_1; M_2; \dots; M_n \text{ end})) \\ (14) \\ &= \text{new } \text{case_res}(\text{TrPI}_{\text{exp}}(\text{self}, E) \parallel (\text{TrPI}_{\text{match}}(\text{case_res}, M_1) + \dots \\ &\quad + \text{TrPI}_{\text{match}}(\text{case_res}, M_n))) \end{aligned} \quad -(19)$$

In rule(19), one process evaluates the *case head* expression and sends the result along fresh channel *case_res* and this *case_res* is used in evaluating the Matches of *case* expression, in stead of using *self* like in *receive* expression. With rule(19), any expression can be used in *case* head provided that evaluation rule(s) is available for that expression which overcomes the limitations of rule(18).

Let us consider a simple *case* expression:

```

case Status of
  true -> dowork;
  false -> take_rest;
  Y -> Y
end.

```

Using rule (19), it is possible to translate the above case expression efficiently and accurately with existing rules for Matches as follows:

```

TrPIexp(self, case Status of
    true -> dowork;
    false -> take_rest;
    Y -> Y
end.)

```

```
(19)
= new case_res (TrPIexp(self, Status) || TrPIexp(case_res, receive
                                true -> dowork;
                                false -> take_rest;
                                Y -> Y
                                end.))
```

$$(14) \quad = \text{new case_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{Status}) \parallel (\text{TrPI}_{\text{match}}(\text{case_res}, \text{true} \rightarrow \text{dowork}) + \text{TrPI}_{\text{match}}(\text{case_res}, \text{false} \rightarrow \text{take_rest}) + \text{TrPI}_{\text{match}}(\text{case_res}, \text{Y} \rightarrow \text{Y})))$$

```
((6) with res -> case_res)
= new case_res( case_res'<Status>.nil || ( TrPImatch(case_res, true -> dowork) +
      TrPImatch(case res, false -> take rest) + TrPImatch(case res, Y -> Y) ) )
```

```
(15) (17A) ->(4) (6)
= new case_res( case_res'<Status>.nil ||
  (case_res(input_pat1).[input_pat1=true]res'<dowork>.nil +
   case_res(input_pat2).[input_pat2=false]res'<take_rest>.nil +
   case_res(Y). res'<Y>.nil ))
```

4.9. Receive and Case: Handling Non-determinism among Matches

If there is only one matching pattern each time of execution in *receive* or *case* expressions then rule(14) and hence rules (15), (16), (17A) and (19) work accurately. But, if there are more than one matches for each execution then the corresponding

receive/case body evaluation choice would be non-deterministic as we have already applied non-determinism between different choices.

Let us consider the translation result of the example of Section 4.8.2,

```
TrPIexp(self, case Status of
    true -> dowork;
    false -> take_rest;
    Y -> Y
end.)
= new case_res( case_res'<Status>.nil ||
    (case_res(input_pat1).[input_pat1=true]res'<dowork>.nil +
    case_res(input_pat2).[input_pat2=false]res'<take_rest>.nil +
    case_res(Y). res'<Y>.nil ))
```

This translation is not error-free. Let us suppose that atom *false* is bound with the variable *Status* in *case head*. In Erlang, only second Match(false ->take_rest) succeeds as there is an order in matching. But in π -calculus translation, we have used non-determinism between different matches and hence, there is a possibility that variable *Status* can match with 2nd or 3rd matches. To avoid such non-determinism between the matching clauses, we have negated (mismatch) the conditions for the preceding matching clauses so that a clause that comes after another clause in order cannot be selected if the preceding one could have been selected. This can be formally defined with the following rule (19A):

```
TrPIexp(self, case E of
    P1->E1;
    ... ;
    Pn->En
end )
:=new case_res(TrPIexp(self, E) ||
    (case_res(input_pat1).[input_pat1= TrPIarg(P1)] TrPIexp(self, E1) +
    case_res(input_pat2).[input_pat2 != TrPIarg(P1)] [input_pat2= TrPIarg(P2)]
    TrPIexp(self, E2) + ....+
    case_res(input_patn).[input_patn != TrPIarg(P1)]... [input_patn != TrPIarg(Pn-1)]
    [input_patn= TrPIarg(Pn)] TrPIexp(self, En) ))
```

-(19A)

If a variable is used as a pattern in any of the matches, then we will use the techniques of rule (17A) and also apply negation(Mismatch) on the patterns of the preceding

clauses. For instance, let pattern P_i is a variable X for i^{th} Match. The translation of this Match will be as follows:

$\text{case_res}(X).[X \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [X \neq \text{TrPI}_{\text{arg}}(P_{i-1})] \text{TrPI}_{\text{exp}}(\text{self}, E_i))$

In this way, we can modify the rules for matches (rules (15) (16) & (17A)) with the help of an additional information, the index (position in order) of the potential Match to be translated in Erlang program.

If an *atom* is used as a pattern in the i^{th} Match, then rule (15), can be modified as follows:

$\text{TrPI}_{\text{match}i}(\text{self}, a \rightarrow E_i)$
 $:= \text{self}(\text{input_pati}).[\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_{i-1})]$
 $[\text{input_pati} = a] \text{TrPI}_{\text{exp}}(\text{self}, E_i)$ -(15A)

Similarly, if a *number* is used as a pattern in the i^{th} Match, then rule (16), can be modified as follows:

$\text{TrPI}_{\text{match}i}(\text{self}, n \rightarrow E_i)$
 $:= \text{self}(\text{input_pati}).[\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_{i-1})]$
 $[\text{input_pati} = \text{unknown}] \text{TrPI}_{\text{exp}}(\text{self}, E_i)$ -(16A)

Finally, if a *variable* is used as a pattern in the i^{th} Match, then rule (17A), can be modified as follows:

$\text{TrPI}_{\text{match}i}(\text{self}, X \rightarrow E_i)$
 $:= \text{self}(X).[X \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [X \neq \text{TrPI}_{\text{arg}}(P_{i-1})] \text{TrPI}_{\text{exp}}(\text{self}, E_i)$ -(17B)

As we see, rule(19A) is a result of applying rules (15A), (16A) and (17B) on rule (19).

As a result, we can now rewrite rule(14) and (19) to be suitable for applying new match rules (15A), (16A) and (17B) by providing the index of the Matches as follows:

Rule(14) can be rewritten as follows:

$\text{TrPI}_{\text{exp}}(\text{self}, \text{receive } M_1; \dots; M_n \text{ end})$
 $:= \text{TrPI}_{\text{match}1}(\text{self}, M_1) + \dots + \text{TrPI}_{\text{match}n}(\text{self}, M_n)$ -(14A)

Similarly, rule(19) can be rewritten as follows:

$\text{TrPI}_{\text{exp}}(\text{self}, \text{case } E \text{ of } M_1; \dots; M_n \text{ end}) := \text{new case_res}(\text{TrPI}_{\text{exp}}(\text{self}, E) \parallel$
 $(\text{TrPI}_{\text{match}1}(\text{case_res}, M_1) + \dots + \text{TrPI}_{\text{match}n}(\text{case_res}, M_n)))$ -(19B)

Consider the following *receive* expression:

```

receive
    saturday ->take_rest ;
    2-> go_i2;
    Monday -> Monday
end

```

Using rule(14A), we can translate this expression by avoiding the non-determinism among matches as follows:

```

TrPIexp(self, receive
    saturday ->take_rest ;
    2-> go_i2;
    Monday ->Monday
end )

```

(14A)

$$:= \text{TrPI}_{\text{match1}}(\text{self}, \text{saturday} \rightarrow \text{take_rest}) + \text{TrPI}_{\text{match2}}(\text{self}, 2 \rightarrow \text{go_i2}) + \text{TrPI}_{\text{match3}}(\text{self}, \text{Monday} \rightarrow \text{Monday})$$

Now applying rule(15A) for $\text{TrPI}_{\text{match1}}$, rule(16A) for $\text{TrPI}_{\text{match2}}$ and rule(17B) for $\text{TrPI}_{\text{match3}}$, we obtain,

```

:=self(input_pat1).[input_pat1=saturday] TrPIexp(self, take_rest)
+ self(input_pat2).[input_pat2 != saturday][input_pat2=unknown] TrPIexp(self, go_i2)
+ self(Monday).[Monday != saturday] [Monday != unknown] TrPIexp(self, Monday)

```

(4) (6)

$$:= \text{self}(\text{input_pat1}).[\text{input_pat1}=\text{saturday}] \text{res}'\langle \text{take_rest} \rangle.\underline{\text{nil}}$$

$$+ \text{self}(\text{input_pat2}).[\text{input_pat2} \neq \text{saturday}][\text{input_pat2}=\text{unknown}] \text{res}'\langle \text{go_i2} \rangle.\underline{\text{nil}}$$

$$+ \text{self}(\text{Monday}).[\text{Monday} \neq \text{saturday}] [\text{Monday} \neq \text{unknown}] \text{res}'\langle \text{Monday} \rangle.\underline{\text{nil}}$$

Similarly, consider the example of Section 4.8.2 again,

```

case Status of
    true -> dowork;
    false -> take_rest;
    Y -> Y
end.

```

Using rule(19B), we can translate this expression by avoiding the non-determinism among matches as follows:

$\text{TrPI}_{\text{exp}}(\text{self}, \text{case Status of}$

```

    true -> dowork;
    false -> take_rest;
    Y -> Y
end)

```

(19B)

$:= \text{new case_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{Status}) \parallel (\text{TrPI}_{\text{match1}}(\text{case_res}, \text{true} \rightarrow \text{dowork}) +$
 $\text{TrPI}_{\text{match2}}(\text{case_res}, \text{false} \rightarrow \text{take_rest}) + \text{TrPI}_{\text{match3}}(\text{case_res}, Y \rightarrow Y))$

((6) with $\text{res} \rightarrow \text{case_res}$)

$:= \text{new case_res}(\text{case_res}'\langle \text{Status} \rangle.\underline{\text{nil}} \parallel (\text{TrPI}_{\text{match1}}(\text{case_res}, \text{true} \rightarrow \text{dowork}) +$
 $\text{TrPI}_{\text{match2}}(\text{case_res}, \text{false} \rightarrow \text{take_rest}) + \text{TrPI}_{\text{match3}}(\text{case_res}, Y \rightarrow Y))$

Now applying rule(15A) for $\text{TrPI}_{\text{match1}}$ & $\text{TrPI}_{\text{match2}}$ and rule(17B) for $\text{TrPI}_{\text{match3}}$, we obtain,

$:= \text{new case_res}(\text{case_res}'\langle \text{Status} \rangle.\underline{\text{nil}} \parallel$
 $(\text{case_res}(\text{input_pat1}).[\text{input_pat1} = \text{true}] \text{TrPI}_{\text{exp}}(\text{self}, \text{dowork}) +$
 $\text{case_res}(\text{input_pat2}).[\text{input_pat2} \neq \text{true}] [\text{input_pat2} = \text{false}] \text{TrPI}_{\text{exp}}(\text{self},$
 $\text{take_rest}) + \text{case_res}(Y). [Y \neq \text{true}] [Y \neq \text{false}] \text{TrPI}_{\text{exp}}(\text{self}, Y))$

(4) (6)

$:= \text{new case_res}(\text{case_res}'\langle \text{Status} \rangle.\underline{\text{nil}} \parallel$
 $(\text{case_res}(\text{input_pat1}).[\text{input_pat1} = \text{true}] \text{res}'\langle \text{dowork} \rangle.\underline{\text{nil}} +$
 $\text{case_res}(\text{input_pat2}).[\text{input_pat2} \neq \text{true}] [\text{input_pat2} = \text{false}] \text{res}'\langle \text{take_rest} \rangle.\underline{\text{nil}}$
 $+ \text{case_res}(Y). [Y \neq \text{true}] [Y \neq \text{false}] \text{res}'\langle Y \rangle.\underline{\text{nil}}))$

4.10 Function Definition

In PIERlang, the general form of a function definition is presented as follows:

Function Definition $F ::= f(X_1, X_2, \dots, X_n) \rightarrow E$
 where $n \geq 0$ and X_1, \dots, X_n are variables.

Each function definition in PIERlang will be translated to a corresponding process definition in π -calculus which can be formally written as follow:

$\text{TrPI}_{\text{fundef}}: \text{Function Definition} \rightarrow \text{Process Definition}$

We consider that a n-ary function definition of Erlang should be translated to a (n + 1)-ary process definition in π -calculus where the first argument will be *self* which has

been included to indicate the current process, executing the function definition in Erlang. Here are the corresponding π -calculus mapping of n-ary and 0-ary function definitions.

$$\text{TrPI}_{\text{fundef}}(\text{self}, f(X_1, \dots, X_n) \rightarrow E) := f(\text{self}, X_1, \dots, X_n) = \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(20)$$

$$\text{TrPI}_{\text{fundef}}(\text{self}, f() \rightarrow E) := f(\text{self}) = \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(20A)$$

Let us consider an example of a 2-ary function definition:

```
start(Monday_Pid, Friday_Pid) ->
    Monday_Pid ! do_work,
    Friday_Pid ! go_market;
```

The overall meaning of this program segment is intuitive. In the variables `Monday_Pid` and `Friday_Pid`, two different PIDs are assigned somewhere else in the program before calling this function definition. When executing this function definition atoms `do_work` and `go_market` are sent to the corresponding processes.

We can now use rule (20) to get a corresponding π -calculus representation of this function definition as follows:

$$\text{TrPI}_{\text{fundef}}(\text{self}, \text{start}(\text{Monday_Pid}, \text{Friday_Pid}) \rightarrow \text{Monday_Pid ! do_work}, \\ \text{Friday_Pid ! go_market};)$$

(20)

$$:= \text{start}(\text{self}, \text{Monday_Pid}, \text{Friday_Pid}) = \text{TrPI}_{\text{exp}}(\text{self}, \text{Monday_Pid ! do_work}, \\ \text{Friday_Pid ! go_market};)$$

(13)

$$= \text{start}(\text{self}, \text{Monday_Pid}, \text{Friday_Pid}) = \text{new exp1_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{Monday_Pid !} \\ \text{do_work}) \parallel \text{exp1_res}(\text{dummy}). (\text{TrPI}_{\text{exp}}(\text{self}, \text{Friday_Pid ! go_market})))$$

(9) (5) & (3)

$$= \text{start}(\text{self}, \text{Monday_Pid}, \text{Friday_Pid}) = \text{new exp1_res}, \text{exp2_res} \\ (\text{Monday_Pid}' <\text{do_work}> .\underline{\text{nil}} \parallel \text{exp1_res}' <\text{do_work}> .\underline{\text{nil}} \parallel \text{exp1_res}(\text{dummy}). (\\ \text{Friday_Pid}' <\text{go_market}> .\underline{\text{nil}} \parallel \text{exp2_res}' <\text{go_market}> .\underline{\text{nil}}))$$

Again, it has to be noted that *self* is used as an input parameter of the translation function along with the function definition to indicate that this function definition is executed by a process whose PID is *self*.

4.11 PIERlang Program

Until now, we have discussed the translation mappings for each of the syntactic constructs of PIERlang-00 with simple corresponding examples. In this section, we

have presented the way of how to translate a complete PIERlang Program in π -calculus.

We consider a function $\text{TrPI}_{\text{program}}$ takes a PIERlang Program as *Input* and will return a System model in the π -calculus as *Output*. A formal representation is as follows:

$\text{TrPI}_{\text{program}} : \text{Program} \rightarrow \text{System}$

$$\begin{aligned} \text{TrPI}_{\text{program}}(\text{self}, F_1; \dots, F_n, E) := & \text{TrPI}_{\text{fundef}}(\text{self}, F_1), \dots, \text{TrPI}_{\text{fundef}}(\text{self}, F_n), \\ & \text{main()} = \text{new self}(\text{TrPI}_{\text{exp}}(\text{self}, E)) \end{aligned} \quad -(21)$$

PIErang Program is composed of a sequence of function definitions followed by an expression, E where E is supposed to be executed in the Erlang Abstract Machine. Usually E is an 0-ary function that is placed at the beginning of the program and mostly the left hand side of the first function definition. Our translated System will start executing with the $\text{main}()$ function in π -calculus.

4.11.1 PIERlang Program 4.1

Let us consider the following program written in PIERlang-00:

```
sender() -> Receiver_Pid = spawn(receiver, []),
        Receiver_Pid ! hello.

receiver() -> receive
            Y -> Y
        end
```

Program 4.1 A simple sender receiver program

4.11.1(a) Execution in Erlang Compiler

In Program 4.1(or Figure 4.12), it is seen that 0-ary function $\text{sender}()$ is invoking a spawn function with 0-ary function $\text{receiver}()$. The PID generating by the $\text{spawn}(\text{receiver}, [])$ function is then assigned to a variable Receiver_Pid along which the atom hello is sent.

In function $\text{receiver}()$, which should be evaluated by the process with PID assigned to Receiver_Pid , has a receive expression. As variable can match any received pattern and thus, directly produces the same output because on the right hand side same variable is placed as an expression.

According to Erlang's *spawn*, *send* and *receive* behaviors, function *receiver()* is now being processed by a process whose PID is assigned to *Receiver_Pid*. In the same

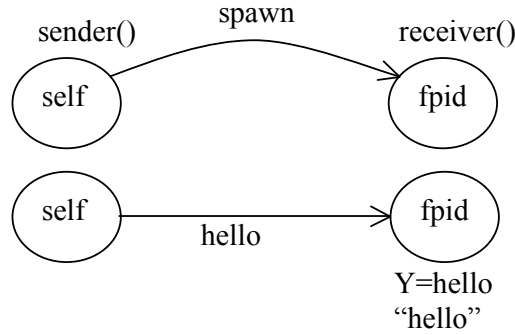


Figure 4.12 Schematic diagram of the simple sender receiver Program 4.1

time, atom *hello* is sent to the function *receiver()* which is the return value of the receiver process(cf. Figure 4.12). In this way, it is found that there is an asynchronous communication between the *sender* and the *receiver* processes.

4.11.1(b) Translation in the π -calculus

According to rule (21) let,

```
F1 ::= sender() ->
    Receiver_Pid = spawn(receiver, []),
    Receiver_Pid ! hello.
```

```
F2 ::= receiver() ->
    receive
        Y -> Y
    end
```

and finally,

```
E ::= sender()
```

Let us start with the translations of the function definitions one by one and then the expression E as follows.

```
TrPIfundef(self, F1) = TrPIfundef(self, sender() ->
    Receiver_Pid = spawn(receiver, []),
    Receiver_Pid ! hello.)
```

```
(20A)
= sender(self) = TrPIexp(self, Receiver_Pid = spawn(receiver, []),
    Receiver_Pid ! hello.)
```

(11)

```
= new rpid, p, receiver_res (p'<rpid>.nil || TrPIexp(rpid, receiver())||  
    receiver_res(dummy).nil || p(Receiver_Pid).TrPIexp(self, Receiver_Pid ! hello) )
```

(10A) & (9)
= new rpid, p, receiver_res, send_res (p' < rpid > .nil || receiver(rpid) ||
receiver_res(dummy).nil || p(Receiver_Pid). ((TrPI_{arg}(**Receiver_Pid**))'
< TrPI_{arg}(**hello**) > .nil || send_res' < TrPI_{arg} (**hello**) > .nil))

```
(3) & (5)
= new rpid, p, receiver_res, send_res (p' < rpid > .nil || receiver(rpid) ||
  receiver_res(dummy).nil || p(Receiver_Pid). (Receiver_Pid' < hello > .nil ||
  send_res' < hello > .nil ))
```

```
TrPIfundef(self, F2) = TrPIfundef(self, receiver() ->
                                receive
                                Y -> Y
                                end )
```

```

(20A)
= receiver(self) = TrPIexp(self, receive
                                Y -> Y
                                end )

```

$$(14) \quad = \text{TrPI}_{\text{match}}(\text{self}, Y \rightarrow Y)$$

```
(17A)
= self(Y).TrPIexp(self, Y)
```

(6)

$$= \text{new receiver } \text{res}(\text{self}(\text{Y}).\text{receiver } \text{res}' < \text{Y} > .\text{nil})$$

and finally,

```
main() = TrPIexp(self, E) = new self(TrPIexp(self, sender() )
```

(10A)

```
= new self (sender(self))
```

It has been observed that we have created a special channel *receiver_res* to store (ready to be received by someone over the same channel) the results received by *receiver()* which is actually the return value of this function. We also notice that we have used the same channel for *spawn* function too as *receiver()* was first started to be executed there.

4.11.1(c) The π -model

From Section 4.11.1(b), the π -model of Program 4.1 can be written as follows:

```
main()=new self (sender(self))

sender(self) = new rpid, p, receiver_res, send_res (p'<rpid>.nil || receiver(rpid) ||
  receiver_res(dummy).nil || p(Receiver_Pid). (Receiver_Pid' <hello>.nil ||
  send_res'< hello > .nil ))

receiver(self) = new receiver_res(self(Y).receiver_res'<Y>.nil)
```

4.11.1(d) Observing Behavior in the π -calculus

To observe the System behavior in π -calculus, we have to start from the *main()* Process.

```
main()=new self(sender(self))

=>new self, rpid, p, receiver_res, send_res (p'<rpid>.nil || receiver(rpid) ||
  receiver_res(dummy).nil || p(Receiver_Pid). (Receiver_Pid' <hello>.nil ||
  send_res'< hello > .nil ))

(react p, Receiver_Pid is bound with PID rpid of receiver process) ->(omitting nil)
=>new self, rpid, p, receiver_res, send_res (receiver(rpid) || receiver_res(dummy).nil
  || (rpid' <hello>.nil || send_res'< hello > .nil ))
```

Message *hello* is sent along *rpid* but no one is here to receive that along channel *rpid*. Therefore, now *receiver(rpid)* has to be instantiated.

```
(substituting rhs definition of receiver(rpid), here self -> rpid )
=>new self, rpid, p, receiver_res, send_res (rpid(Y).receiver_res'<Y>.nil ||
  receiver_res(dummy).nil || (rpid' <hello>.nil || send_res'< hello > .nil ))

(react on rpid, Y is now bound with hello )
=>new self, rpid, p, receiver_res, send_res (receiver_res'<hello>.nil ||
  receiver_res(dummy).nil || (nil || send_res'< hello > .nil ))
```

In π -calculus System, we got the same message *hello* in the result channel of receiver process *receiver_res* which exactly fulfills our expected behavior of translated system in π -calculus.

4.11.2 PIERlang Program 4.2

Let us consider the example of the Finite State Machine (FSM) from [1].

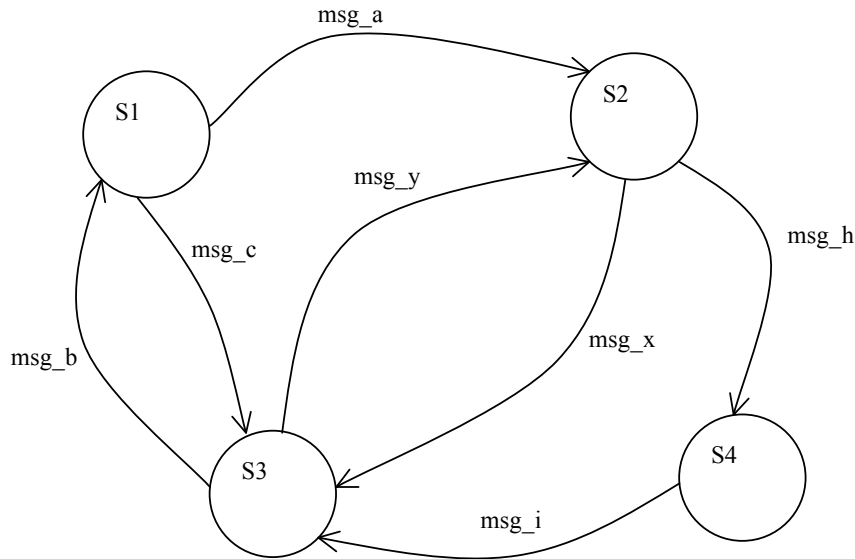


Figure 4.13 Graphical representation of Simple FSM Program 4.2.

The Figure 4.12 shows a simple FSM with four states, the possible transitions and the events which cause them. One easy way to program such a *state X event* machine is shown in Program 4.2. In this code, we are only interested in how to represent the states and manage the transitions between them. Each state is represented by a separate function and events are represented by messages.

```

start()->
    State_Pid = spawn(s1, []),
    State_Pid ! msg_a.
  
```

```

s1()->
    receive
        msg_a-> s2();
        msg_c-> s3()
    end.
  
```

```
s2()->
  receive
    msg_x-> s3();
    msg_h-> s4()
  end.
```

```
s3()->
  receive
    msg_b-> s1();
    msg_y-> s2()
  end.
```

```
s4()->
  receive
    msg_i-> s3()
  end.
```

Program 4.2 A simple FSM program

4.11.2(a) Execution in Erlang Compiler

The state functions (Figure 4.12 and Program 4.2) wait in a *receive* for an event message. When a message has been received, the FSM makes a transition to the new state by calling the function for that state. By making sure that each call to a new state is a last call, the FSM process will evaluate in constant space.

To have a simplified version of event passing, we have only considered that message *msg_a* is sent to state one by the main program. From the diagram or PIERlang program above, it is clear that if *State One* is received a message named *msg_a*, then there would be changes of state, from *State One* to *State Two*. We would like to observe now the corresponding π -calculus System behavior by translating the above program into π -calculus System.

4.11.2(b) Translation in the π -calculus

According to rule (21) let,

```
F1::= start()->
  State_Pid = spawn(s1, []);
  State_Pid ! msg_a;
```

```

F2::= s1()->
  receive
    msg_a-> s2();
    msg_c-> s3();
  end.

```

```

F3::= s2()->
  receive
    msg_x-> s3();
    msg_h-> s4();
  end.

```

```

F4::=s3()->
  receive
    msg_b-> s1();
    msg_y-> s2();
  end.

```

```

F5::= s4()->
  receive
    msg_i-> s3();
  end.

```

```

E::=start()

```

Here,

```

main()=new self(TrPIexp( self, start()))

```

```

(10A)
=new self (start(self))

```

```

TrPIfunder(self, F1):= TrPIfunder(self, start()->
                                                    State_Pid = spawn(s1, []),
                                                    State_Pid ! msg_a. )

```

```

(20A)
=start(self) = TrPIexp(self, State_Pid = spawn(s1, [])
                      State_Pid ! msg_a. )

```

```

(11)
=new fpid, receiver_res, p ( p'<fpid>.nil || TrPIexp(fpid, s1()) ||
  receiver_res(dummy).nil || p(State_Pid).(TrPIexp(self, State_Pid ! msg_a ) ) )

```

(10A) & (9) \rightarrow (5) & (3)

```
=new fpid, receiver_res, p, send_res ( p'<fpid>.nil || s1(fpid) ||
  receiver_res(dummy).nil || p(State_Pid).( State_Pid'<msg_a>.nil ||
  send_res'<msg_a>.nil ))
```

TrPI_{fundef}(self, F₂): = TrPI_{fundef}(self, s1()) \rightarrow

```
  receive
    msg_a-> s2();
    msg_c-> s3();
  end.)
```

(20A)

```
= s1(self)= TrPIexp(self, receive
  msg_a-> s2();
  msg_c-> s3();
end.)
```

(14)

= TrPI_{match}(self, **msg_a**-> **s2()**) + TrPI_{match}(self, **msg_c**-> **s3()**)

(15)

= self(input_pat1).[input_pat1 = msg_a] TrPI_{exp}(self, **s2()**) +
 self(input_pat2).[input_pat2 = msg_c] TrPI_{exp}(self, **s3()**)

(10A)

= self(input_pat1).[input_pat1 = msg_a] s2(self) +
 self(input_pat2).[input_pat2 = msg_c] s3(self)

Similarly,

TrPI_{fundef}(self, F₃): = TrPI_{fundef}(self, s2()) \rightarrow

```
  receive
    msg_x-> s3();
    msg_h-> s4();
  end.)
```

(20A) \rightarrow (14) \rightarrow (15) \rightarrow (10A)

=s2(self)= self(input_pat1).[input_pat1 = msg_x] s3(self) +
 self(input_pat2).[input_pat2 = msg_h] s4(self)

TrPI_{fundef}(self, F₄): = TrPI_{fundef}(self, s3()) \rightarrow

```
  receive
    msg_b-> s1();
    msg_y-> s2();
  end.)
```

(20A) $\rightarrow (14) \rightarrow (15) \rightarrow (10A)$
 $= s3(self) = self(input_pat1).[input_pat1 = msg_b] s1(self) +$
 $self(input_pat2).[input_pat2 = msg_y] s2(self)$

$TrPI_{fundef}(self, F_5) := TrPI_{fundef}(self, s4()) \rightarrow$
 $receive$
 $msg_i \rightarrow s3();$
 $end.)$

(20A) $\rightarrow (15) \rightarrow (10A)$
 $= s4(self) = self(input_pat).[input_pat = msg_i] s3(self)$

4.11.2(c) The π -model

From Section 4.11.2(b), the π -model of Program 4.2 can be written as follows:

$main() = \underline{new} self (start(self))$
 $start(self) = \underline{new} fpid, receiver_res, p, send_res (p' < fpid > .\underline{nil} \parallel s1(fpid) \parallel$
 $receiver_res(dummy).\underline{nil} \parallel p(State_Pid).(State_Pid' < msg_a > .\underline{nil} \parallel$
 $send_res' < msg_a > .\underline{nil}))$
 $s1(self) = self(input_pat1).[input_pat1 = msg_a] s2(self) +$
 $self(input_pat2).[input_pat2 = msg_c] s3(self)$
 $s2(self) = self(input_pat1).[input_pat1 = msg_x] s3(self) +$
 $self(input_pat2).[input_pat2 = msg_h] s4(self)$
 $s3(self) = self(input_pat1).[input_pat1 = msg_b] s1(self) +$
 $self(input_pat2).[input_pat2 = msg_y] s2(self)$
 $s4(self) = self(input_pat).[input_pat = msg_i] s3(self)$

4.11.2(d) Observing Behavior in the π -calculus

To observe the System behavior in π -calculus, we have to start from the *main()* Process.

$main() = \underline{new} self (start(self))$
(substituting rhs definition of **start(self)**)
 $= \underline{new} self, fpid, receiver_res, p, send_res (p' < fpid > .\underline{nil} \parallel s1(fpid) \parallel$
 $receiver_res(dummy).\underline{nil} \parallel p(State_Pid).(State_Pid' < msg_a > .\underline{nil} \parallel$
 $send_res' < msg_a > .\underline{nil}))$

(react on p , $State_pid$ is now bound with PID $fpid$)->(Omitting \underline{nil} process)
 $\Rightarrow \underline{new}$ self, $fpid$, receiver_res, p , send_res ($s1(fpid) \parallel receiver_res(dummy).\underline{nil} \parallel$
 $(fpid' <msg_a>.\underline{nil} \parallel send_res' <msg_a>.\underline{nil})$)

(substituting rhs definition of $s1(self)$ with $self \rightarrow fpid$)
 $\Rightarrow \underline{new}$ self, $fpid$, receiver_res, p , send_res (($fpid(input_pat1).[input_pat1 =$
 $msg_a] s2(fpid) + fpid(input_pat2).[input_pat2 = msg_c] s3(fpid)) \parallel$
 $receiver_res(dummy).\underline{nil} \parallel (fpid' <msg_a>.\underline{nil} \parallel send_res' <msg_a>.\underline{nil})$)

(react on $fpid$, $input_pat1$ is now bound with atom msg_a)
 $\Rightarrow \underline{new}$ self, $fpid$, receiver_res, p , send_res (($[msg_a = msg_a] s2(fpid)) \parallel$
 $receiver_res(dummy).\underline{nil} \parallel (\underline{nil} \parallel send_res' <msg_a>.\underline{nil})$)

A name matching $[msg_a = msg_a]$ is found and in this way we move from *State One* to *State Two* by means of calling the corresponding process calls which is same as we can expect from PIERlang Program 4.2 and from the FSM diagram in Figure 4.12.

4.11.2(e) An Enriched FSM of Program 4.2

Now consider that $F1$ is enriched with one more *send* expression as follows:

```
F1 ::= start()->
    State_Pid = spawn(s1, []);
    State_Pid ! msg_a;
    State_Pid ! msg_h;
```

Initially, *spawn* function is called with *State One* and then messages msg_a and msg_h are sent sequentially. According to the FSM diagram or its PIERlang representation above, we should now move from *State One* to *State two* for message msg_a and then from *State Two* to *State Four* for message msg_h . First, we have translated this enriched *start()* function and then we have tried to observe what could be happened in π -calculus with this enriched system. Other translations remain same as of Section 4.11.2(c).

```
TrPIfundef(self, F1) := TrPIfundef(self, start()->
    State_Pid = spawn(s1, []);
    State_Pid ! msg_a,
    State_Pid ! msg_h.)
```

(20A)

```
=start(self) = TrPIexp(self, State_Pid = spawn(s1, []);
                    State_Pid ! msg_a,
                    State_Pid ! msg_h.)
```

(11)

```
=new fpid, receiver_res, p( p'<fpid>.nil || TrPIexp(fpid, s1()) ||
    receiver_res(dummy).nil || p(State_Pid).(TrPIexp(self, State_Pid ! msg_a ,
    State_Pid ! msg_h ) ) )
```

(10A) & (13)

```
=new fpid, receiver_res, p, send1_res( p'<fpid>.nil || s1(fpid) ||
    receiver_res(dummy).nil || p(State_Pid).(TrPIexp(self, State_Pid ! msg_a) ||
    send1_res(dummy).( TrPIexp(self, State_Pid ! msg_h ) ) )
```

(9) ->(5) & (3)

```
=new fpid, receiver_res, p, send1_res, send2_res( p'<fpid>.nil || s1(fpid) ||
    receiver_res(dummy).nil || p(State_Pid).( State_Pid'< msg_a>.nil ||
    send1_res'<msg_a>.nil || send1_res(dummy).( State_Pid'<msg_h>.nil ||
    send2_res'<msg_h>.nil ) ) )
```

Observing behavior in π -calculus:

```
main()=new self(start(self))
```

(substituting rhs definition of **start(self)**)

```
=>new self, fpid, receiver_res, p, send1_res, send2_res( p'<fpid>.nil || s1(fpid) ||
    receiver_res(dummy).nil || p(State_Pid).( State_Pid'< msg_a>.nil ||
    send1_res'<msg_a>.nil || send1_res(dummy).( State_Pid'<msg_h>.nil ||
    send2_res'<msg_h>.nil ) ) )
```

(react on **p**, **State_pid** is now bound with PID **fpid**) ->(Omitting **nil** process)

```
=>new self, fpid, receiver_res, p, send1_res, send2_res(s1(fpid) ||
    receiver_res(dummy).nil || (fpid'< msg_a>.nil || send1_res'<msg_a>.nil ||
    send1_res(dummy).( fpid'<msg_h>.nil || send2_res'<msg_h>.nil ) ) )
```

(substituting rhs definition of **s1(self)** with **self -> fpid**)

```
=>new self, fpid, receiver_res, p, send1_res, send2_res( (
    fpid(input_pat1).[input_pat1=msg_a] s2(fpid) + fpid(input_pat2).[input_pat2=
    msg_c] s3(fpid) ) || receiver_res(dummy).nil || (fpid'< msg_a>.nil ||
    send1_res'<msg_a>.nil || send1_res(dummy).( fpid'<msg_h>.nil ||
    send2_res'<msg_h>.nil ) ) )
```

(react on *fpid*, *input_pat1* is now bound with atom *msg_a*)
 \Rightarrow new self, fpid, receiver_res, p, send1_res, send2_res(([msg_a=msg_a] s2(fpid))
 || receiver_res(dummy).nil || (nil || send1_res'<msg_a>.nil ||
 send1_res(dummy).(fpid'<msg_h>.nil || send2_res'<msg_h>.nil)))

Here a name matching [msg_a=msg_a] is found and thereby moving to *State Two*.

(substituting rhs definition of s2(self) with self -> fpid, react on send1_res, this is a dummy reaction to maintain the sequence of evaluation)
 \Rightarrow new self, fpid, receiver_res, p, send1_res, send2_res((fpid(input_pat1).[input_pat1=msg_x] s3(fpid) + fpid(input_pat2).[input_pat2=msg_h] s4(fpid)) ||
 receiver_res(dummy).nil || (nil || nil || (fpid'<msg_h>.nil ||
 send2_res'<msg_h>.nil)))

(react on *fpid*, *input_pat2* is now bound with atom *msg_h*)
 \Rightarrow new self, fpid, receiver_res, p, send1_res, send2_res(([msg_h=msg_h] s4(fpid))
 || receiver_res(dummy).nil || (nil || nil || (nil || send2_res'<msg_h>.nil)))

Again a name matching [msg_h=msg_h] is found and we see that with this name matching we are moving to *State Four* which is the same as we can expect from the PIERlang Program code and FSM diagram above.

4.12 TrPIs at a Glance

In this section we have presented the translation mapping rules in a top-down fashion so that any user can easily grasp the rules while translation mapping. Although there are more rules in the corresponding sections, we have only presented those rules here that we will use frequently in translation mappings.

4.12.1 Frequently Used TrPIs

This section consists of most frequently used translation rules.

TrPI_{program} : Program -> System

$$\text{TrPI}_{\text{program}}(\text{self}, F_1; \dots, F_n, E) := \text{TrPI}_{\text{fundef}}(\text{self}, F_1), \dots, \text{TrPI}_{\text{fundef}}(\text{self}, F_n), \\ \text{main}() = \text{new self}(\text{TrPI}_{\text{exp}}(\text{self}, E)) \quad -(21)$$

TrPI_{fundef} : Function Definition -> Process Definition

$$\text{TrPI}_{\text{fundef}}(\text{self}, f(X_1, \dots, X_n) \rightarrow E) := f(\text{self}, X_1, \dots, X_n) = \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(20)$$

$$\text{TrPI}_{\text{fundef}}(\text{self}, f() \rightarrow E) := f(\text{self}) = \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(20A)$$

TrPI_{exp}: Name X Expression -> Process

$$\text{TrPI}_{\text{exp}}(\text{self}, n) := \text{res}'\langle \text{unknown} \rangle . \underline{\text{nil}} \quad -(2)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, a) := \text{res}'\langle a \rangle . \underline{\text{nil}} \quad -(4)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, X) := \text{res}'\langle X \rangle . \underline{\text{nil}} \quad -(6)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, X = E_1, E_2) := \underline{\text{new}} \text{exp1_res}(\text{TrPI}_{\text{exp}}(\text{self}, E_1) \parallel \text{exp1_res}(X). \text{TrPI}_{\text{exp}}(\text{self}, E_2)) \quad -(7)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, X = E) := \text{TrPI}_{\text{exp}}(\text{self}, E) \parallel \text{res}(X). \underline{\text{nil}} \quad -(8)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, A_1! A_2) := (\text{TrPI}_{\text{arg}}(A_1))' \langle \text{TrPI}_{\text{arg}}(A_2) \rangle . \underline{\text{nil}} \parallel \text{res}'\langle \text{TrPI}_{\text{arg}}(A_2) \rangle . \underline{\text{nil}} \quad -(9)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, f(A_1, A_2, \dots, A_n)) := f(\text{self}, \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n)) \quad -(10)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, f()) := f(\text{self}) \quad -(10A)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, X = \text{spawn}(f, [A_1, \dots, A_n]), E) &:= \underline{\text{new}} \text{fpid}, p, f_res(p' \langle \text{fpid} \rangle . \underline{\text{nil}} \parallel \\ &\quad \text{TrPI}_{\text{exp}}(\text{fpid}, f(A_1, \dots, A_n)) \parallel f_res(\text{dummy}). \underline{\text{nil}} \parallel p(X). \text{TrPI}_{\text{exp}}(\text{self}, E)) \quad -(11) \end{aligned}$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{spawn}(f, [A_1, \dots, A_n]), E) &:= \underline{\text{new}} \text{fpid}, f_res(\text{TrPI}_{\text{exp}}(\text{fpid}, f(A_1, \dots, \\ &\quad A_n)) \parallel f_res(\text{dummy}). \underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{self}, E)) \quad -(12) \end{aligned}$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, E_1, E_2) &:= \underline{\text{new}} \text{exp1_res}(\text{TrPI}_{\text{exp}}(\text{self}, E_1) \parallel (\text{exp1_res}(\text{dummy}). \\ &\quad \text{TrPI}_{\text{exp}}(\text{self}, E_2))) \quad -(13) \end{aligned}$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{receive } M_1; \dots; M_n \text{ end}) \\ := \text{TrPI}_{\text{match}}(\text{self}, M_1) + \dots + \text{TrPI}_{\text{match}}(\text{self}, M_n) \quad -(14) \end{aligned}$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{case } E \text{ of } M_1; \dots; M_n \text{ end}) \\ := \underline{\text{new}} \text{case_res}(\text{TrPI}_{\text{exp}}(\text{self}, E) \parallel (\text{TrPI}_{\text{match}}(\text{case_res}, M_1) + \dots \\ + \text{TrPI}_{\text{match}}(\text{case_res}, M_n))) \quad -(19) \end{aligned}$$

TrPI_{match}: Name X Match -> Process

$$\text{TrPI}_{\text{match}}(\text{self}, a \rightarrow E) := \text{self}(\text{input_pat}).[\text{input_pat} = a] \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(15)$$

$$\text{TrPI}_{\text{match}}(\text{self}, n \rightarrow E) := \text{self}(\text{input_pat}).[\text{input_pat} = \text{unknown}] \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(16)$$

$$\text{TrPI}_{\text{match}}(\text{self}, X \rightarrow E) := \text{self}(X). \text{TrPI}_{\text{exp}}(\text{self}, E) \quad -(17A)$$

TrPI_{arg}: Argument -> Name

$$\text{TrPI}_{\text{arg}}(n) := \text{unknown} \quad -(1)$$

$$\text{TrPI}_{\text{arg}}(a) := a \quad -(3)$$

$$\text{TrPI}_{\text{arg}}(X) := X \quad -(5)$$

4.12.2 TrPIs for Handling Non-determinism among Matches

This section consists of the rules for handling non-determinism among the Matches of *receive* and *case* expressions.

TrPI_{exp}: Name X Expression -> Process

$$\begin{aligned} &\text{TrPI}_{\text{exp}}(\text{self}, \text{receive } M_1; \dots; M_n \text{ end}) \\ &:= \text{TrPI}_{\text{match1}}(\text{self}, M_1) + \dots + \text{TrPI}_{\text{matchn}}(\text{self}, M_n) \end{aligned} \quad -(14A)$$

$$\begin{aligned} &\text{TrPI}_{\text{exp}}(\text{self}, \text{case } E \text{ of } M_1; \dots; M_n \text{ end}) := \text{new case_res}(\text{TrPI}_{\text{exp}}(\text{self}, E) \parallel \\ &\quad (\text{TrPI}_{\text{match1}}(\text{case_res}, M_1) + \dots + \text{TrPI}_{\text{matchn}}(\text{case_res}, M_n))) \end{aligned} \quad -(19B)$$

TrPI_{match}: Name X Match -> Process

$$\begin{aligned} &\text{TrPI}_{\text{matchi}}(\text{self}, a \rightarrow E_i) \\ &:= \text{self}(\text{input_pati}).[\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_{i-1})] \\ &\quad [\text{input_pati} = a] \text{TrPI}_{\text{exp}}(\text{self}, E_i) \end{aligned} \quad -(15A)$$


$$\begin{aligned} &\text{TrPI}_{\text{matchi}}(\text{self}, n \rightarrow E_i) \\ &:= \text{self}(\text{input_pati}).[\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [\text{input_pati} \neq \text{TrPI}_{\text{arg}}(P_{i-1})] \\ &\quad [\text{input_pati} = \text{unknown}] \text{TrPI}_{\text{exp}}(\text{self}, E_i) \end{aligned} \quad -(16A)$$

$$\begin{aligned} &\text{TrPI}_{\text{matchi}}(\text{self}, X \rightarrow E_i) \\ &:= \text{self}(X).[X \neq \text{TrPI}_{\text{arg}}(P_1)] \dots [X \neq \text{TrPI}_{\text{arg}}(P_{i-1})] \text{TrPI}_{\text{exp}}(\text{self}, E_i) \end{aligned} \quad -(17B)$$

Chapter 5

Mapping Tuples with Polyadic Communications

This chapter focuses on the use of tuple in Erlang with its corresponding translation mapping in π -calculus using polyadic communication. The asynchronous polyadic π -calculus is used to provide a detailed step by step translation of any non-nested tuple-based expression of Erlang. Several distinguishable PIERlang programs are translated to π -calculus system models and compared the execution behavior with their corresponding PIERlang programs. This chapter also gives a formal treatment of the BIF *self()*.

Table of Contents 	5.1 PIERlang-01 Syntax	-69
	5.2 BIF self()	-70
	5.2.1 BIF self():Used As Argument	-71
	5.2.2 PIERlang Program 5.1: self() as Argument	-71
	5.2.2(a) Execution in PIERlang Compiler	-71
	5.2.2(b) Translation in the π -calculus	-72
	5.2.2(c) The π -Model	-72
	5.2.2(d) Observing Behavior in π -calculus	-72
	5.2.3 BIF self(): Used As Expression	-73
	5.2.4 PIERlang Program 5.2: self() as Expression	-73
	5.2.4(a) Execution in PIERlang Compiler	-73
	5.2.4(b) Translation in the π -calculus	-74
	5.2.4(c) The π -Model	-75
	5.2.4(d) Observing Behavior in π -calculus	-75
	5.3 Tuples as Expression	-76
	5.4 Tuples in Send Expression: PID as Implicit Mailbox	-76
	5.5 Tuples in Receive Expression: PID as Implicit Mailbox	-77
	5.5.1 Receive Action in Polyadic π -calculus	-77
	5.5.2 Matches of Receive Expression	-78
	5.5.3 Matches	-79
	5.5.3(a) Matches: Atoms & Numbers as Elements of Tuple	-80
	5.5.3(b) Matches: Variables as Elements of Tuple	-80
	5.5.4 PIERlang Program 5.3: CAR for DC Rule	-82
	5.5.4(a) Execution in PIERlang Compiler	-82
	5.5.4(b) Translation in the π -calculus	-83
	5.5.4(c) The π -Model	-84
	5.5.4(d) Observing Behavior in π -calculus: CAR for DC	-85
	5.5.5 PIERlang Program 5.4: Variables as Tuple Elements	-89
	5.5.5(a) Execution in PIERlang Compiler	-89
	5.5.5(b) Translation in the π -calculus	-90
	5.5.5(c) The π -Model	-91
	5.5.5(d) Observing Behavior in π -calculus	-91

5.5.6 PIERlang Program 5.5: Inaccuracy of Rule(26A)	-92
5.5.6(a) Execution in PIERlang Compiler	-93
5.5.6(b) Translation in the π -calculus	-94
5.5.6(c) The π -Model	-95
5.5.6(d) Observing Behavior in π -calculus: Rule (26A) is Insufficient.	-95
5.5.7 An Approach to Improve Rule (26A)	-96
5.5.7(a) Modification of Rule (26A): Providing BVS	-97
5.5.7(b) Modification of Rule (26B): Providing BNS	-98
5.5.7(c) Betterment of Rule (26C) over Rule(26B)	-100
5.5.8 PIERlang Program 5.5: Rule (26C) is Sound but has Extra Names in BNS	-100
5.5.8(a) Execution in PIERlang Compiler	-100
5.5.8(b) Translation in the π -calculus	-100
5.5.8(c) The π -Model	-103
5.5.8(d) Observing Behavior in π -calculus	-104
5.5.9 Improving Rule (26C): Providing BNSRAV	-108
5.5.10 PIERlang Program 5.6: Rule (26D) Sounds Perfect	-109
5.5.10(a) Execution in PIERlang Compiler	-110
5.5.10(b) Translation in the π -calculus	-110
5.5.10(c) The π -Model	-113
5.5.10(d) Observing Behavior in π -calculus	-114
5.6 Tuples in Case Expressions	-119
5.6.1 PIERlang Program 5.7: Tuples in Case Expression	-119
5.6.1(a) Execution in PIERlang Compiler	-119
5.6.1(b) Translation in the π -calculus	-120
5.6.1(c) The π -Model	-121
5.6.1(d) Observing Behavior in π -calculus	-122
5.7 An Approach to Improve Send Rule (25)	-123
5.7.1 PIERlang Program 5.8: Send rule(25) is Insufficient	-124
5.7.1(a) Execution in PIERlang Compiler	-124
5.7.1(b) Translation in the π -calculus	-125
5.7.1(c) The π -Model	-126
5.7.1(d) Observing Behavior in π -calculus	-126
5.7.2 A Modification to Send Rule (25)	-127
5.7.3 PIERlang Program 5.8: Send Rule(25A) Sounds Correct	-127
5.7.3(a) Execution in PIERlang Compiler	-127
5.7.3(b) Translation in the π -calculus	-128
5.7.3(c) The π -Model	-128
5.7.3(d) Observing Behavior in π -calculus	-128
5.8 An Approach to Improve Tuple Expression Rule (24)	-130
5.9 An Alternative Approach for Match Rule (26D)	-130
5.10 TrPIs at a Glance	-131

5.1 PIERlang-01 Syntax

In Chapter 4, we have discussed the translation mapping based on the PIERlang-00, which was really a very restricted subset of Erlang. In this chapter, we will present a

wider PIERlang Version named as PIERlang-01 with its corresponding translation mapping in the π -calculus. This version of PIERlang is a composition of the previous version PIERlang-00 and some additional syntactic constructs marked as boldface as in the following Figure 5.1 for supporting use of tuples as a message in *send* expression and as *patterns* in matches of *receive* and *case* expressions. In this Chapter 5, along with the new rules, we have also used the translation mapping rules (rule (1) to rule (21)) presented in Chapter 4 and therefore, in this Chapter first rule will be numbered with rule (22), second will be with (23) and so on.

Program	$P ::= F+; E$	
Function Definition	$F ::= f(X_1, X_2, \dots, X_n) \rightarrow E$	$; n \geq 0$
Expression	$E ::= n \mid a \mid X$	
	$\mid X = E_1, E_2 \mid X = E \mid E_1, E_2$	
	$\mid f(A_1, \dots, A_n) \mid \mathbf{self}()$	$; n \geq 0$
	$\mid \text{spawn}(f, [A_1, A_2, \dots, A_n])$	$; n \geq 0$
	$\mid \{\mathbf{A_1}, \dots, \mathbf{A_n}\}$	$; n \geq 0$
	$\mid A_1 ! A_2 \mid \mathbf{A} ! \{\mathbf{A_1}, \dots, \mathbf{A_n}\}$	$; n \geq 0$
	$\mid \text{receive } \mathbf{M_1}; \dots; \mathbf{M_n} \text{ end}$	$; n > 0$
	$\mid \text{case } \mathbf{E} \text{ of } \mathbf{M_1}; \dots; \mathbf{M_n} \text{ end}$	$; n > 0$
Match	$M ::= P \rightarrow E \mid \{\mathbf{P_1}, \dots, \mathbf{P_n}\} \rightarrow \mathbf{E}$	$; n \geq 0$
Pattern	$P ::= n \mid a \mid X$	
Argument	$A ::= n \mid a \mid X \mid \mathbf{self}()$	
$n \in \text{Numbers(Integer \& Float)};$		
$a, f \in \text{Atoms};$		
$X, X_1, \dots, X_n \in \text{Variables};$		

Figure 5.1 PIERlang-01 (added syntactic constructs are marked with **boldface**)

Many of the core concepts discussed in this chapter would resemble the concepts that we have already discussed in Chapter 4. Here our intention is to incorporate uses of *tuples* in *send*, *receive* and *case* expressions applying the same concept that we have discussed in Chapter 4 for number, atom and variable. We have also introduced the built-in-function *self()* of Erlang.

5.2 BIF self()

First addition to PIERlang-01 is *self()*, a BIF of Erlang which is frequently used in Erlang to represent current Process Identifier(PID) of the process executing the

expression. We have found that this BIF can also be used as an argument for *function calls*, *send*, *receive* and *case* expressions.

A process in the PIERlang or even in Erlang has the more or less same behavior(cf. Section 1.2) as in the π -calculus; therefore, a PID in Erlang would be a unique process in π -calculus. At that moment, we have only considered the *self()* function of our PIERlang. It has two types of translation depending on the context where it is used.

5.2.1 BIF *self()*: Used As Argument

If *self()* is used as an argument for function call, send expression, receive expression or for case expression then the translation would be as follow:

$$\text{TrPI}_{\text{arg}}(\text{self}()) := \text{self} \quad \text{-(22)}$$

Here *self* would not be a new name in the π -calculus. It should be noted that we have used *self* for translation mapping in Chapter 4. It is used there as a new name within the whole system indicating that the expression where it is used, is executed by a process which has a PID *self*. As we have already considered *self* as a new name in our translation in the π -calculus, we will not consider it as a separate new name anymore.

5.2.2 PIERlang Program 5.1: *self()* as Argument

Let us consider a simple example[11] where *self* is used as an argument of the *send* expression:

```
foo() ->
    self() ! stop,
    receive
        stop -> terminate
    end.
```

Program 5.1. A program that sends a message to itself and then terminates.

5.2.2(a) Execution in PIERlang Compiler

Program 5.1 is a very simple program that sends a message to itself and then terminates. Function *foo()* is executed by a process and the PID of that process is represented by the BIF *self()* which implies that *foo* process sends atom *stop* to its own. On the other hand, it waits to receive any atomic message. If the received message is an atom *stop*, it will be terminated with atom *terminate*.

5.2.2(b) Translation in the π -calculus

According to rule (21), here, $\text{main}() = \text{new self}(\text{TrPI}_{\text{exp}}(\text{self}, \text{foo}()))$

(10A)
 $= \text{new self}(\text{foo}(\text{self}))$

$\text{TrPI}_{\text{funder}}(\text{self}, \text{foo}() \rightarrow$
 $\text{self}() ! \text{stop},$
 receive
 $\text{stop} \rightarrow \text{terminate}$
 end.)

(20A) \rightarrow (13)
 $= \text{foo}(\text{self}) = \text{new send_res}(\text{TrPI}_{\text{exp}}(\text{self}, \text{self}() ! \text{stop}) \parallel \text{send_res}(\text{dummy}).($
 $\text{TrPI}_{\text{exp}}(\text{self}, \text{receive}$
 $\text{stop} \rightarrow \text{terminate}$
 end)))

((9) \rightarrow (22) & (3)) & (15)
 $= \text{new send_res}(\text{self}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}(\text{dummy}).($
 $\text{self}(\text{input_pat}).[\text{input_pat}=\text{stop}] \text{TrPI}_{\text{exp}}(\text{self}, \text{terminate})))$

(4, new result channel **receive_res** is added to send result of the receiver)
 $= \text{new send_res}(\text{self}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}(\text{dummy}).($
 $\text{self}(\text{input_pat}).[\text{input_pat}=\text{stop}] \text{receive_res}'\langle \text{terminate} \rangle . \underline{\text{nil}}))$

5.2.2(c) The π -Model

From Section 5.2.2(b), the π -model of Program 5.1 can be written as follows:

$\text{main}() = \text{new self}(\text{foo}(\text{self}))$
 $\text{foo}(\text{self}) = \text{new send_res}(\text{self}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}'\langle \text{stop} \rangle . \underline{\text{nil}} \parallel \text{send_res}(\text{dummy}).($
 $\text{self}(\text{input_pat}).[\text{input_pat}=\text{stop}] \text{receive_res}'\langle \text{terminate} \rangle . \underline{\text{nil}}))$

5.2.2(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

$\text{main}() = \text{new self}(\text{foo}(\text{self}))$

(substituting RHS of process **foo(self)**)

```
=>new self, send_res (self'<stop>.nil || send_res'<stop>.nil || send_res(dummy).(
    self(input_pat).[input_pat=stop] receive_res'<terminate>.nil ) )
```

(react on *send_res*, dummy reaction rule)

```
=>new self, send_res (self'<stop>.nil || nil || ( self(input_pat).[input_pat=stop]
    receive_res'<terminate>.nil ) )
```

(react on *self*, *input_pat* is bound with *stop*)

```
=>new self, send_res ( nil || nil || ( [stop=stop] receive_res'<terminate>.nil ) )
```

A name matching [*stop*=*stop*] is found and consequently atom *terminate* is sent along *receive_res* which is the same intention of the PIERlang Program 5.1(cf. Section 5.2.2(a)).

5.2.3 BIF self(): Used As Expression

The translation of *self()* would be different when it is used as an expression. In that case, it is assumed that the value of *self()* will be passed through the so called result channel *res*, the same way we have used in the translation mapping for variables, atoms or numbers as expressions in Section 4.2.

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{self}()) := \text{res}'\langle\text{TrPI}_{\text{arg}}(\text{self}())\rangle.\text{nil}$$

(22)

$$= \text{res}'\langle\text{self}\rangle.\text{nil} \quad \text{-(23)}$$

5.2.4 PIERlang Program 5.2: self() as Expression

Let us consider the same program as of Program 5.1 where *self()* is used as an expression as in the following way:

```
foo() ->
    MyID=self(),
    MyID ! stop,
    receive
        stop -> terminate
    end.
```

Program 5.2. A program that sends a message to itself and then terminates.

5.2.4(a) Execution in PIERlang Compiler

The execution behavior of Program 5.2 same as Program 5.1 described in Section 5.2.2(a). Only difference here is that *self()* is used as an expression. The value of *self()* is first assigned to a variable *MyID* and the message *stop* sends to the process that has its PID assigned to *MyID*.

5.2.4(b) Translation in the π -calculus

According to rule (21),

`main() = new self(foo(self))` (as Program 5.1)

Here *self* is a global name within this π -calculus system.

```
TrPIfundef(self, foo() ->
    MyID=self(),
    MyID ! stop,
    receive
    stop -> terminate
end.)
```

```
(20A)
=foo(self)= TrPIexp(self,
    MyID=self(),
    MyID ! stop,
    receive
    stop -> terminate
end. )
```

(8, new name ***assn_res*** is added for assignment expression)
`=new assn_res (TrPIexp(self, self()) || assn_res(MyID).(TrPIexp(self,
MyID ! stop,
 receive
stop -> terminate
 end.)))`

(23) (13, new name ***send_res*** is added)
`=new assn_res, send_res(assn_res'<self>.nil || assn_res(MyID).(TrPIexp(self,
 MyID ! stop) || send_res(dummy).(TrPIexp(self,
 receive
stop -> terminate
 end.))))`

((9) -> (5) (3)) & ((15) -> (3))
`=new assn_res, send_res(assn_res'<self>.nil || assn_res(MyID).(MyID'<stop>.nil ||
 sene_res'<stop>.nil || send_res(dummy).(self(input_pat).[input_pat=stop]
 TrPIexp(self, terminate))))`

(4, new result channel ***receive_res*** is added to send result of the receiver)
`=new assn_res, send_res, receive_res(assn_res'<self>.nil || assn_res(MyID).(`

```
MyID'<stop>.nil || sene_res'<stop>.nil || send_res(dummy).(
self(input_pat).[input_pat=stop] receive_res'<terminate>.nil ) ) )
```

5.2.4(c) The π -Model

From Section 5.2.4(b), the π -model of Program 5.2 can be written as follows:

```
main() = new self(foo(self))

foo(self) = new assn_res, send_res, receive_res( assn_res'<self>.nil || assn_res(MyID).(
  MyID'<stop>.nil || sene_res'<stop>.nil || send_res(dummy).(
    self(input_pat).[input_pat=stop] receive_res'<terminate>.nil ) ) ) )
```

5.2.4(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main() = new self(foo(self))

(substituting RHS of process foo(self))
= new self, assn_res, send_res, receive_res( assn_res'<self>.nil || assn_res(MyID).(
  MyID'<stop>.nil || sene_res'<stop>.nil || send_res(dummy).(
    self(input_pat).[input_pat=stop] receive_res'<terminate>.nil ) ) ) )

(react on assn_res, this communication binds MyID with self)
=> new self, assn_res, send_res, receive_res(nil || (self'<stop>.nil || sene_res'<stop>.nil
  || send_res(dummy).(self(input_pat).[input_pat=stop] receive_res'<terminate>.nil )))

=>(cf. 5.2.2(d) Observing behavior in the  $\pi$ -calculus)
```

Program 5.2 is interesting mainly for two reasons. First reason is that we have presented here how *self()* could be used as an expression. We have also tried to present the similarities between the *self* used as translation function parameter and *self* used as expression. It has been found that both are actually expressing the same thing, the current process identifier of the process executing the expression or function. Another reason behind using this example is to show how 3 different types of expressions (assignment, send and receive) are evaluated sequentially during the translation mapping.

5.3 Tuples as Expression

Second extension to PIERlang-01 is the uses of tuple as an expression. We found that tuples are not translated uniformly. Tuples are control structure for grouping variables and values and therefore, an intrinsic part of the expressions in which they are used. According to our added tuple syntax only numbers, atoms or variables could be used as tuple elements. We have found that we could use the Polyadic π -calculus to evaluate such tuple expression as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, \{A_1, A_2, \dots, A_n\}) = \text{res}' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} \quad -(24)$$

Here it has been seen that tuple elements are translated using the TrPI_{arg} function and are sent as names along the so called *res* channel in polyadic form.

5.4 Tuples in Send Expression: PID as Implicit Mailbox

Our third addition to PIERlang-01 is the uses of tuples as a message in *send* expression. The added send expression syntax is $A ! \{A_1, \dots, A_n\}$. From this syntax, it is clear that our new *send* expression can send a non-nested tuple of number(s), atom(s), variable(s) and *self()* as message to a process's mailbox, which has a PID identified by *A*.

We have also found that Polyadic π -calculus could be used to model such tuple-based *send* expression as follows:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, A ! \{A_1, A_2, \dots, A_n\}) = & (\text{TrPI}_{\text{arg}}(A))' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} \\ & \parallel \text{res}' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} \end{aligned} \quad -(25)$$

Translation of such a *send* expression has introduced two subprocesses working in parallel; one is a direct mapping of the *send* expression in the Polyadic π -calculus and another sends the message $\{A_1, \dots, A_n\}$ to the *res* channel so that any other process waiting for a message along *res* channel can receive the message $\{A_1, \dots, A_n\}$ with a *receive* expression along *res*. In this way, the message is sent to the specific process identified by the PID *A* and through the *res* channel. Of course, it is required to call the $\text{TrPI}_{\text{arg}}()$ function to have a corresponding π -calculus translation of the elements of the tuple message of the *send* expression.

Let us consider the following example,

```
DestPid ! {dowork, Take_rest, 5, self()}
```

In Erlang, this send expression means that a tuple of atom *dowork*, variable *Take_rest* number 5 and the process identifier of the current process *self()* are sent to the mailbox of the process identified by the PID stored in the variable *DestPid*.

Using rule (25), we can obtain a corresponding Polyadic π -calculus representation of the above expression as follows:

$$\begin{aligned}
 & \text{TrPI}_{\text{exp}}(\text{self}, \text{DestPid} ! \{\text{dowork}, \text{Take_rest}, 5, \text{self}()\}) \\
 (25) \quad & := ((\text{TrPI}_{\text{arg}}(\text{DestPid}))' < \text{TrPI}_{\text{arg}}(\text{dowork}), \text{TrPI}_{\text{arg}}(\text{Take_rest}), \text{TrPI}_{\text{arg}}(5), \\
 & \quad \text{TrPI}_{\text{arg}}(\text{self}()) > .\underline{\text{nil}} \parallel \text{res}' < \text{TrPI}_{\text{arg}}(\text{dowork}), \text{TrPI}_{\text{arg}}(\text{Take_rest}), \text{TrPI}_{\text{arg}}(5), \\
 & \quad \text{TrPI}_{\text{arg}}(\text{self}()) > .\underline{\text{nil}}) \\
 (5) (3) (1) \& (22) \\
 & = \text{DestPid}' < \text{dowork}, \text{Take_rest}, \text{unknown}, \text{self} > .\underline{\text{nil}} \\
 & \quad \parallel \text{res}' < \text{dowork}, \text{Take_rest}, \text{unknown}, \text{self} > .\underline{\text{nil}}
 \end{aligned}$$

Here in the Polyadic π -calculus representation, it is seen that the argument translations of atom *dowork*, variable *Take_rest*, number 5 and PID *self()* are sent as names along the channel *DestPid* and *res*. In this way, channel *DestPid* is used as an implicit mailbox of a process whose PID is *DestPid*. A detailed discussion can be found about modelling PID as the implicit mailbox in Sections 4.4 and 4.7.

5.5 Tuples in Receive Expression: PID as Implicit Mailbox

We have discussed the sending a tuple of message to the destination process by considering the PID of the process to which the message is sent as the channel along which the message is to be sent. In this way, any process can receive the message along the channel that resembles its PID.

5.5.1 Receive Action in Polyadic π -calculus

Let us consider the example of Section 5.4. Here message $\{\text{dowork}, \text{Take_rest}, 5, \text{self}()\}$ is sent to a process whose PID is *DestPid*. We have done this by sending the message along channel *DestPid* in Polyadic π -calculus. Now consider, this process with PID *DestPid* intends to receive that message and after receiving that message let us consider, it would be subprocess Q. This receiving scenario can be represented in Polyadic π -calculus as follows:

$$\text{DestPid}(\text{input_pat1}, \text{input_pat2}, \text{input_pat3}, \text{input_pat4}).Q$$

Here bound names *input_pat1*, *input_pat2*, *input_pat3* and *input_pat4* will be bound to Q with the corresponding receiving elements along channel *DestPid*. With the *react* rule of π -calculus, we can represent this scenario. We consider that the final translated

result of the example of Section 5.4 and current receiving process work in parallel, which can be formally written as follows:

```

DestPid'<dowork, Take_rest, unknown, self>.nil || res'<dowork,
  Take_rest, unknown, self>.nil || DestPid(input_pat1, input_pat2, input_pat3,
    input_pat4).Q

(react on DestPid)
=>nil || res'<dowork, Take_rest, unknown, self>.nil || Q[input_pat1/dowork,
  input_pat2/Take_rest, input_pat3/unknown, input_pat4/self,]

```

Process Q will now behave with the sent message along *DestPid*. In Section 5.4, we have implicitly modelled the mailbox of Erlang by considering the PID of a process as the mailbox of the process to which the intended message is sent. We have done this in π -calculus by considering the PID as the channel and sending the intended message along that channel. Here (Section 5.5.1), the sent message is received along the same channel name(PID of the process) and thus modelling mailbox implicitly. A detailed discussion can be found about modelling PID as the implicit mailbox in Sections 4.4 and 4.7.

5.5.2 Matches of Receive Expression

Until now, we have discussed so far the sending and basic receiving mechanism of PIERlang in Polyadic π -calculus perspective. The sending of message discussed in Section 5.4 is sound enough to meet the presented *send* expression. The receiving mechanism in PIERlang-01 can be described more generally as follows:

Let us consider an arbitrary *receive* expression that could be written in PIERlang-01.

```

receive
  {P1, P2, ..., P10}-> Body1;
  {P11, P12}-> Body2;
  P13-> Body3;
  {P14, P15, P16}-> Body4
end

```

In this *receive* expression, the patterns of the matches are either variable length tuples or a single pattern. Whatever the patterns could be this *receive* expression resembles the general syntax of *receive* expression of PIERlang-00 syntax,

```

receive

```



```

Pattern1 -> Body1;
...;
PatternN -> BodyN
end

```

For this general *receive syntax* we have already presented translation mapping rule (14). We will use rule(14) for translation mapping of the proposed tuple-based *receive* expression too. Using rule(14),

```

TrPIexp(self,
  receive
    {P1, P2, ..., P10}-> Body1;
    {P11, P12}-> Body2;
    P13-> Body3;
    {P14, P15, P16}-> Body4
  end )
(14)
:=TrPImatch(self, {P1, P2, ..., P10}-> Body1) + TrPImatch(self, {P11, P12}-> Body2) +
  TrPImatch(self, P13-> Body3) + TrPImatch(self, {P14, P15, P16}-> Body4)

```

We will consider that TrPI_{match} will now capable of handling tuple of patterns in the Matches of receive expression.

5.5.3 Matches

We have already discussed the intention behind rule(14) in section 4.7. We have also discussed about the translation mapping of matches of single pattern match using rules (15) (16) & (17A) in Section 4.7 where the potential received messages could be a number, an atom or a variable. In this section, the intention is to use the tuples of number(s), atom(s), variable(s) and/or self() as the message of the *send* expression (cf. Section 5.4) in *receive* expression. According to the presented *send* expression (cf. section 5.4), a match in the *receive* expression should be $\{P_1, \dots, P_n\} \rightarrow E$ where P_1, \dots, P_n could be a number, an atom or a variable.

5.5.3(a) Matches: Atoms & Numbers as Elements of Tuple Message

We have already discussed how a tuple of message could implicitly be sent to the mailbox of a process in Section 5.4 above. Now the intention is to receive the message from the implicit mailbox (actually from a channel whose name is same as the PID of

the receiver process) of the process. We have discussed how a tuple of message can be received along a channel using Polyadic π -calculus in Section 5.5.1. One important issue left to discuss is matching of the received message(s) against the patterns. We handle this pattern matching for a match of tuple pattern using the name matching feature of Polyadic π -calculus. We can receive the tuple of message(s) using the *receive* statement translation rule in Polyadic π -calculus along a channel (say self) where the message elements will be bound to the corresponding input names and then applying the name matching feature on the bound names against the corresponding patterns of the *receive* statement. Of course, we have to use TrPI_{arg} to get a corresponding translation of the patterns(elements of tuple pattern) before applying name matching. This general scenario can be formally presented as with the following rule(26) for numbers and atoms as the elements of the tuple pattern:

$$\begin{aligned} & \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) \\ &= \text{self}(\text{input_pat1}, \dots, \text{input_patn}).[\text{input_pat1}=\text{TrPI}_{\text{arg}}(P_1)][\text{input_pat2}= \\ & \quad \text{TrPI}_{\text{arg}}(P_2)] \dots [\text{input_patn}=\text{TrPI}_{\text{arg}}(P_n)]\text{TrPI}_{\text{exp}}(\text{self}, E). \end{aligned} \quad -(26)$$

Where $P_1, \dots, P_n \in \{\text{Numbers}, \text{Atoms}\}$.

5.5.3(b) Matches: Variables as Elements of Tuple Message

A variable can match with any term, therefore, *no name matching* is required for variable(s) as pattern(s). In this way, if we have variable(s) in P_1, \dots, P_n , we will omit the name matching for that corresponding pattern. This variable pattern could be used in the *Body expression E* where the corresponding received message is expected to be substituted in place of the variable. Having this in mind, the substitution feature of the π -calculus is used on the corresponding bound name(s). A detailed discussion can be found in Section 4.7.2(c) using monadic π -calculus.

Let us consider that patterns P_2 and P_3 are variables (say X and Y) in the above rule (26), then the corresponding translation formula would be as follows:

$$\begin{aligned} & \text{TrPI}_{\text{match}}(\text{self}, \{P_1, X, Y, \dots, P_n\} \rightarrow E) \\ &= \text{self}(\text{input_pat1}, \dots, \text{input_patn}).[\text{input_pat1}=\text{TrPI}_{\text{arg}}(P_1)][\text{input_pat4}= \\ & \quad \text{TrPI}_{\text{arg}}(P_4)] \dots [\text{input_patn}=\text{TrPI}_{\text{arg}}(P_n)](\text{TrPI}_{\text{exp}}(\text{self}, E)[X/\text{input_pat2}, \\ & \quad Y/\text{input_pat3}]) \end{aligned}$$

Where $[Z/\text{input_pati}]$ denotes the replacement of every free occurrence of Z in translated E by input_pati .

A more simplified version can be obtained as follows:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, X, Y, \dots, P_n\} \rightarrow E) \\ = \text{self}(\text{input_pat1}, X, Y, \dots, \text{input_patn}).[\text{input_pat1} = \text{TrPI}_{\text{arg}}(P_1)][\text{input_pat4} = \\ \text{TrPI}_{\text{arg}}(P_4)] \dots [\text{input_patn} = \text{TrPI}_{\text{arg}}(P_n)] \text{TrPI}_{\text{exp}}(\text{self}, E) \end{aligned}$$

Here it has been noticed that we have directly used the pattern variables X and Y in the *receive* action and we have omitted the substitutions that we have done above for variable patterns with corresponding bound names of the *receive* action. We have also omitted the corresponding name matching where $\text{variable}(s)$ is used as $\text{pattern}(s)$.

We know a variable can be bound with any term, therefore, we used the same pattern variable(s) as the bound name(s) in *receive action*. In this way, pattern variable(s) will be bound with the corresponding received message(s). Now if this variable is used in expression E , we will have no problem since we know a variable can be bound only once and the pattern variable(s) has already been bound with the corresponding message by the *receive action*. This general scenario can be formally represented with rule(26A) which supports numbers, atoms and variables as pattern elements of the tuple pattern:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is} \\ \text{a variable } X. \end{array} \right. \quad \text{-(26A)} \end{aligned}$$

We can now get the final translation for a tuple-based Match with rule (26A) by combining all the cases of P_i where $i \in \{1, \dots, n\}$.

5.5.4 PIERlang Program 5.3: CAR for DC

Let us consider the following simple example[31] written in PIERlang-01.

```
ping() ->
  Pong_ID = spawn(pong, []),
```

```

Pong_ID ! {self(), ping},
receive
  pong -> pong
end.

pong() ->
receive
  {Ping_ID, ping} -> Ping_ID ! pong
end.

```

Program 5.3 A simple ping-pong program.

5.5.4(a) Execution in PIErlang Compiler

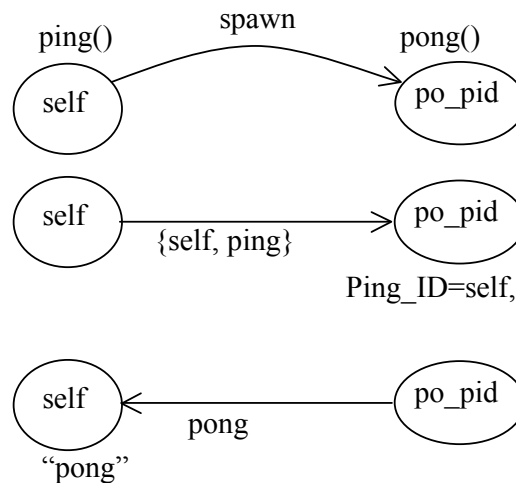


Figure 5.2 Schematic diagram of Program 5.3

In Program 5.3 (or in Figure 5.2), there are two functions *ping* and *pong* where *ping* is the starting function. Function *ping* has a *spawn* call function which initiates the execution of the *pong* function (or we can say pong process) in parallel with ping process. This *spawn* call also returns the PID (po_pid) of *pong* process in variable *Pong_ID*. The 2nd line of *ping* process sends the atom *ping* along with its identify (PID self) as a tuple of message to *pong* process.

Ping process also waits to receive an atom *pong* and if matching successful, it will return the atom *pong* as well. Pong process (pong function) waits to receive a tuple of message where the first element of the tuple is a variable and 2nd one is an atom *ping*.

As it is already mentioned a variable can be bound with any term, therefore, matching of the *receive* would be successful if the 2nd element of the received message is an atom *ping*. It has been already seen that ping process is willing to send a tuple of message to pong process where the 2nd element of the message is an atom *ping*, therefore, pattern matching will be successful and the pattern variable *Ping_ID* will be bound with the sent message element *self()* where *self()* is the process identity of the *ping* process. As a result, atom *pong* will be sent to the mailbox of the ping process, which has been done here by the expression *Ping_ID ! pong*, where variable *Ping_ID* contains the process identity of ping process. In this way, processes ping and pong will communicate and execute in parallel by sending messages to each other.

5.5.4(b) Translation in the π -calculus

We will now translate Program 5.3 in Polyadic π -calculus and will verify whether it is possible or not to get the same behavior in π -calculus as per Section 5.5.4(a).

According to rule (21), $\text{main}() = \text{new self (ping(self))}$

$\text{TrPI}_{\text{funder}}(\text{self, ping()}) \rightarrow$

```

    Pong_ID = spawn(pong, []),
    Pong_ID ! {self(), ping},
    receive
    pong -> pong
end. )

```

(20A) \rightarrow (11, we have considered *po_pid* as pong PID)
 $= \text{ping(self)} = \text{new po_pid, p, pong_res}(p' < \text{po_pid} > . \underline{\text{nil}} \parallel \text{TrPI}_{\text{exp}}(\text{po_pid, pong()}) \parallel$
 $\text{pong_res(dummy).} \underline{\text{nil}} \parallel \text{p(Pong_ID). (TrPI}_{\text{exp}}(\text{self,}$
 $\text{Pong_ID ! \{self(), ping\},}$
 receive
 pong -> pong
 $\text{end.)))$

(10A) (13, new name *pong_send_res* is added)
 $= \text{new po_pid, p, pong_res, pong_send_res (p' < po_pid > .} \underline{\text{nil}} \parallel \text{pong(po_pid)} \parallel$
 $\text{pong_res(dummy).} \underline{\text{nil}} \parallel \text{p(Pong_ID). (TrPI}_{\text{exp}}(\text{self, Pong_ID ! \{self(), ping\}}) \parallel$
 $\text{pong_send_res(dummy). (TrPI}_{\text{exp}}(\text{self receive}$

```

    pong -> pong
end. ) ) )

```

```

((25) -> (5) (22) (3)) & ((15)->(4, new name ping_res is added ))
= new po_pid, p, pong_res, pong_send_res, ping_res ( p'<po_pid>.nil ||
    pong(po_pid)|| pong_res(dummy).nil || p(Pong_ID). (Pong_ID'<self, ping>.nil ||
    pong_send_res'<self, pong>.nil || pong_send_res(dummy). (
    self(input_pat).[input_pat=pong] ping_res'<pong>.nil ) ) )

```

```

TrPIfunder(self, pong() ->
    receive
        {Ping_ID, ping} -> Ping_ID ! pong
    end. )

```

```

(20A)
=pong(self)=TrPIexp(self,
    receive
        {Ping_ID, ping} -> Ping_ID ! pong
    end. )

```

```

(26A)
= self(Ping_ID, input_pat).[input_pat=ping]TrPIexp(self, Ping_ID ! pong)

```

```

(9, pong_res is used to send final result of pong process, its already a name in the
whole system)->(5) (3)
= self(Ping_ID, input_pat).[input_pat=ping] ( Ping_ID'<pong>.nil ||
    pong_res'<pong>.nil )

```

5.5.4(c) The π -Model

From Section 5.5.4(b), the π -model of Program 5.3 can be written as follows:

```

main()=new self (ping(self))

```

```

ping(self)= new po_pid, p, pong_res, pong_send_res, ping_res ( p'<po_pid>.nil ||
    pong(po_pid) || pong_res(dummy).nil || p(Pong_ID). (Pong_ID'<self, ping>.nil ||
    pong_send_res'<self, pong>.nil || pong_send_res(dummy). (
    self(input_pat).[input_pat=pong] ping_res'<pong>.nil ) ) )

```

```
pong(self)= self(Ping_ID, input_pat).[input_pat=ping] ( Ping_ID'<pong>.nil ||
    pong_res'<pong>.nil )
```

5.5.4(d) Observing Behavior in π -calculus: CAR for DC Rule

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self (ping(self))
```

(substituting RHS of process *ping(self)*)

```
=>new self, po_pid, p, pong_res, pong_send_res, ping_res (
    p'<po_pid>.nil || pong(po_pid) || pong_res(dummy).nil || p(Pong_ID).
    (Pong_ID'<self, ping>.nil || pong_send_res'<self, pong>.nil ||
    pong_send_res(dummy). ( self(input_pat).[input_pat=pong] ping_res'<pong>.nil )))
```

PID of pong process(*po_pid*) is sent over channel *p* and at the same time another process is ready to receive that PID along *p* and then binds with *Pong_ID*. In this way, the PID of pong process is bound with *Pong_ID*.

(react on *p*, *Pong_ID* is now bound with pong PID, *po_pid*) -> (Omitting nil process)

```
=>new self, po_pid, p, pong_res, pong_send_res, ping_res (
    // Pong Process
    pong(po_pid) || pong_res(dummy).nil ||

    // Ping Process
    ( po_pid'<self, ping>.nil || pong_send_res'<self, pong>.nil ||
    pong_send_res(dummy). (self(input_pat).[input_pat=pong] ping_res'<pong>.nil )))
```

Tuple of message $\{self, ping\}$ is sent over channel *po_pid* but there is no process to receive this message. In the same time, ping process can receive a message along *self* channel to have a pattern matching with the received message but no one sends message over *self* channel. However, *pong(self)* process call can be instantiated with its PID *po_pid* as follows:

(Substituting RHS definition of *pong(self)* with *self* -> *po_pid*)

```
=>new self, po_pid, p, pong_res, pong_send_res, ping_res (
    // Pong Process
    po_pid(Ping_ID, input_pat).[input_pat=ping] ( Ping_ID'<pong>.nil ||
    pong_res'<pong>.nil ) || pong_res(dummy).nil ||
```

```
// Ping Process
( po_pid'<self, ping>.nil || pong_send_res'<self, pong>.nil ||
  pong_send_res(dummy). (self(input_pat).[input_pat=pong] ping_res'<pong>.nil )))
```

At that moment, tuple of message $\{self, ping\}$ is sent along po_pid and there is also a receiver subprocess along po_pid in ping process, thereby with the *react* rule of π -calculus tuple of message can be received and in this way $Ping_ID$ will be bound with $self$ and $input_pat$ with $ping$.

```
(react on po_pid)
=>new self, po_pid, p, pong_res, pong_send_res, ping_res (
  // Pong Process
  [ping=ping] (self'<pong>.nil || pong_res'<pong>.nil ) || pong_res(dummy).nil ||

  // Ping Process
  ( nil || pong_send_res'<self, pong>.nil || pong_send_res(dummy). (
    self(input_pat).[input_pat=pong] ping_res'<pong>.nil )))
```

After applying reaction rule on po_pid , $input_pat$ has been replaced with $ping$ and $Ping_ID$ with $self$ and consequently, a name matching $[ping=ping]$ is found. As name matching is successful, atom $pong$ is sent over channel $self$. The same behavior is found from the Program 5.3 in Section 5.5.4(a). In Program 5.3, as soon as ping process receives a tuple of message $\{Ping_ID, ping\}$, its pattern matching will be successful and atom $pong$ will be sent to the process whose PID is $Ping_ID$. $Ping_ID$ will be bound with a PID during pattern matching. Now let's see whether we can get the same behavior here or not.

In the system state above, it has been found that although atom $pong$ is sent (by pong process) along channel $self$ (ping process) as a result of successful name matching $[ping=ping]$, there is no direct process that can receive that atom $pong$ along $self$ channel. However, we have found that there is one process (here ping process) has the possibility of receiving something along $self$ channel but before doing so, it has to perform a *dummy* receive operation along $pong_send_res$. This dummy *receive* action confirms that ping process (here PID $self$) will not receive anything until it can send the tuple of message $\{self, ping\}$ to pong process, thereby maintaining the sequence of evaluating the expressions (send and receive expressions in ping process). This dummy receive action also indicates that pong process will receive the tuple of message $\{self, ping\}$ and then it will send an atom $pong$ to ping process. Ping process

can only receive atom *pong* when *pong* process sent it along *self*. In this way, the execution mechanism of PIERlang programs are implemented in π -calculus translated models too.

We can apply *react* rule along channel *pong_send_res*. But there is still some confusion due to the mixing of Polyadic and Monadic π -calculus while translating the PIERlang code. The code segment related to channel *pong_send_res* is:

```
pong_send_res'<self, pong>.nil || pong_send_res(dummy).(  
  self(input_pat).[input_pat=pong] ping_res'<pong>.nil )
```

The sender side sends a tuple of size 2 but the receiving process can only receive an atomic message. As a result, there will be run time error in π -calculus system execution. We have tried to solve this problem by considering that whenever a message is sent over a certain channel and there is another process ready to receive message over the same channel for dummy communications, we will suppose that receiver channel has the same semantic meaning as the sender one and we will change our receiver channel semantic accordingly. Thus, in the example code above, we will change the receiver channel semantic to adapt it with the sender channel semantic as follows by replacing monadic input action with polyadic one:

```
pong_send_res'<self, pong>.nil || pong_send_res(dummy1, dummy2).(  
  self(input_pat).[input_pat=pong] ping_res'<pong>.nil )
```

The sender side channel, *pong_send_res* sends a tuple of size 2, therefore, the receiver channel semantic should be changed in such a way that it can receive a tuple of size 2 also. To meet this goal, we have changed the receiver side channel semantic from Monadic π -calculus to Polyadic π -calculus, thus enabling it to receive a tuple of size 2. We call this π -calculus rule as Channel Adaptation Rule for Dummy Communications (CAR for DC). This rule will not be applicable to other normal communication(s). Here is a formal representation of *CAR for DC*:

$\text{res}' \langle \text{name1}, \text{name2}, \dots, \text{namen} \rangle . \underline{\text{nil}} \parallel \text{res}(\text{dummy}).R$

(CAR for DC on res)

$\Rightarrow \text{res}' \langle \text{name1}, \text{name2}, \dots, \text{namen} \rangle . \underline{\text{nil}} \parallel \text{res}(\text{dummy1}, \text{dummy2}, \dots, \text{dummyn}).R$

(react on *res*)

$\Rightarrow \underline{\text{nil}} \parallel R$

Thereby, we can apply rule (CAR for DC) on the above π -calculus system as follows:

(CAR for DC on pong_send_res)

$\Rightarrow \text{new self, po_pid, p, pong_res, pong_send_res, ping_res (}$

// Pong Process

$[\text{ping}=\text{ping}] (\text{self}' \langle \text{pong} \rangle . \underline{\text{nil}} \parallel \text{pong_res}' \langle \text{pong} \rangle . \underline{\text{nil}}) \parallel \text{pong_res}(\text{dummy}). \underline{\text{nil}} \parallel$

// Ping Process

$(\underline{\text{nil}} \parallel \text{pong_send_res}' \langle \text{self, pong} \rangle . \underline{\text{nil}} \parallel \text{pong_send_res}(\text{dummy1}, \text{dummy2}). ($
 $\text{self}(\text{input_pat}).[\text{input_pat}=\text{pong}] \text{ping_res}' \langle \text{pong} \rangle . \underline{\text{nil}})))$

(react on *pong_send_res*, dummy communication to maintain sequence of execution as in PIERlang semantic)

$\Rightarrow \text{new self, po_pid, p, pong_res, pong_send_res, ping_res (}$

// Pong Process

$[\text{ping}=\text{ping}] (\text{self}' \langle \text{pong} \rangle . \underline{\text{nil}} \parallel \text{pong_res}' \langle \text{pong} \rangle . \underline{\text{nil}}) \parallel \text{pong_res}(\text{dummy}). \underline{\text{nil}} \parallel$

// Ping Process

$(\underline{\text{nil}} \parallel \underline{\text{nil}} \parallel (\text{self}(\text{input_pat}).[\text{input_pat}=\text{pong}] \text{ping_res}' \langle \text{pong} \rangle . \underline{\text{nil}})))$

Now there is a possibility of communication along channel *self*.

(react on *self, input_pat* is bound with atom *pong*)

$\Rightarrow \text{new self, po_pid, p, pong_res, pong_send_res, ping_res (}$

// Pong Process

$(\underline{\text{nil}} \parallel \text{pong_res}' \langle \text{pong} \rangle . \underline{\text{nil}}) \parallel \text{pong_res}(\text{dummy}). \underline{\text{nil}} \parallel$

// Ping Process

$(\underline{\text{nil}} \parallel \underline{\text{nil}} \parallel ([\text{pong}=\text{pong}] \text{ping_res}' \langle \text{pong} \rangle . \underline{\text{nil}})))$

As a result of this communication, a name matching $[\text{pong}=\text{pong}]$ is successful and consequently, atom *pong* is now sent along *ping_res* channel meeting the PIERlang behavior according to our expectation described in Section 5.5.4(a).

5.5.5 PIERlang Program 5.4: Variables as Tuple Elements

Let us consider a simple echo process [1] where both the pattern in the *receive* expression are variables.

```

start() ->
    Loop_Pid=spawn(loop, []),
    Loop_Pid ! {self(), hello}.

loop() ->
    receive
        {From, Message} ->
            From ! Message,
            loop()
    end.

```

Program 5.4 A simple echo process

5.5.5(a) Execution in PIERlang Compiler

The execution mechanism of this simple echo process is shown as schematic diagram in Figure 5.3.

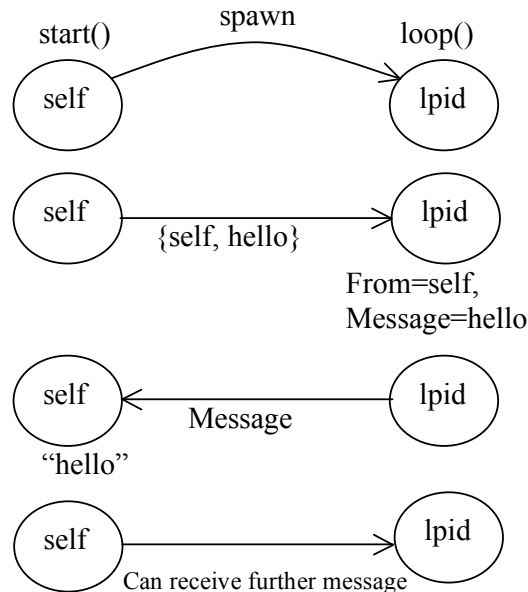


Figure 5.3 Schematic diagram of the simple echo process of Program 5.4.

Here start process with PID *self* invokes a spawn call for loop process with process identifier *lpid*. At the same time, start process sends the tuple of message $\{self, hello\}$ to the loop process. In loop process both patterns are variables and hence are matched with the sent message. In this way, variables *From* and *Message* are bound with *self* and *hello* respectively. Consequently, atom *hello* is sent back to the start process. Loop process can be also restarted again to receive further message(s).

5.5.5(b) Translation in the π -calculus

According to rule (21),

```
main()=new self(start(self))
```

```
TrPIfunder(self, start() ->
    Loop_Pid=spawn(loop, []),
    Loop_Pid ! {self(), hello})
```

```
(20A)->(11)->(25) (10A) (22) (5) (3)
=start(self) = new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid).( Loop_Pid'<self, hello>.nil ||
    start_send_res'<self, hello>.nil ) )
```

```
TrPIfunder(self, loop() ->
    receive
        {From, Message} ->
            From ! Message,
            loop()
    end. )
```

```
(20A) ->(26A)->(13)
=loop(self) = new loop_send_res ( self(From, Message). ( TrPIexp(self, From !
    Message) || loop_send_res(dummy).TrPIexp(self, loop()) ) )
```

```
(9) ->(5) (10A)
= new loop_send_res ( self(From, Message). (From'<Message>.nil ||
    loop_send_res'<Message>.nil || loop_send_res(dummy). loop(self) ) )
```

5.5.5(c) The π -Model

From Section 5.5.5(b), the π -model of Program 5.4 can be written as follows:

```
main()=new self(start(self))

start(self) = new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
  loop_res(dummy).nil || p(Loop_Pid).( Loop_Pid'<self, hello>.nil ||
  start_send_res'<self, hello>.nil ) )

loop(self) = new loop_send_res ( self(From, Message). (From'<Message>.nil ||
  loop_send_res'<Message>.nil || loop_send_res(dummy). loop(self) ) )
```

5.5.5(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))

(substituting RHS of process start(self))
=>new self, lpid, p, loop_res, start_send_res(
  p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil || p(Loop_Pid).(
  Loop_Pid'<self, hello>.nil || start_send_res'<self, hello>.nil ) )

(react on p, Loop_Pid is now bound with lpid) ->(Omitting nil process)
=>new self, lpid, p, loop_res, start_send_res(
  // loop process
  loop(lpid) || loop_res(dummy).nil ||
  // start process
  (lpid'<self, hello>.nil || start_send_res'<self, hello>.nil ) )

(substituting RHS of process loop(self) with lpid as parameter i.e. self -> lpid)
=>new self, lpid, p, loop_res, start_send_res, loop_send_res (
  // loop process
  ( lpid(From, Message). (From'<Message>.nil ||
  loop_send_res'<Message>.nil || loop_send_res(dummy).loop(lpid) ) )
  || loop_res(dummy).nil ||
  // start process
  (lpid'<self, hello>.nil || start_send_res'<self, hello>.nil ) )
```

```

(react on lpid, From is bound with self and Message is with hello )
=>new self, lpid, p, loop_res, start_send_res, loop_send_res (
    // loop process
    ( (self'<hello>.nil || loop_send_res'<hello>.nil || loop_send_res(dummy).
      loop(lpid) ) ) || loop_res(dummy).nil ||
    // start process
    ( nil || start_send_res'<self, hello>.nil ) )

```

Now it is clear that message *hello* is again sent back to the start process. It has been done here by *self'<hello>.nil*. We know PID of start process is *self*. Therefore, start process can receive message *hello* with a receive action along channel *self*.

However, there is also a possibility for the loop process to be initiated again to receive further message(s), which is done here with *loop(lpid)*. But it is required to perform a dummy communications on channel *loop_send_res* before doing so. The purpose of this dummy communication is to force the loop process to start a new session after sending the received message to the sender (here start process), thus maintaining the sequence of execution as Figure 5.3(cf. Section 5.5.5(a)).

```

(react on loop_send_res)
=>new self, lpid, p, loop_res, start_send_res, loop_send_res (
    // loop process
    ( (self'<hello>.nil || nil || loop(lpid) ) ) || loop_res(dummy).nil ||
    // start process
    ( nil || start_send_res'<self, hello>.nil ) )

```

It is now possible to start a new session of message receiving by the loop process with *loop(lpid)* which semantically same as Program 5.4 or in Figure 5.3.

5.5.6 PIERlang Program 5.5: Inaccuracy of Rule (26A)

Until now, rule (26A) is found correct. In this section, the accuracy of rule(26A) is investigated and found that it is not a sufficient rule for handling variable patterns in receive statement Matches.

Let us consider Program 5.5 which is also an echo process[1] with some additional code segments. Here start process(start function) creates a simple echo process (loop process) which returns any message sent to it. The expression *spawn(loop, [])* causes the function *loop()* (loop process) to be evaluated in parallel with the calling function *start()* (start process).

```

start() ->
    Loop_Pid = spawn(loop, []),
    Loop_Pid ! {self(), hello},
    receive
        { Loop_Pid, Msg} -> Msg
    end,
    Loop_Pid ! stop.

```

```

loop() ->
    receive
        {Main_Pid, Msg} ->
            Main_Pid ! {self(), Msg},
            loop();
    stop ->
        true
    end.

```

Program 5.5 An echo process

5.5.6(a) Execution in PIERlang Compiler

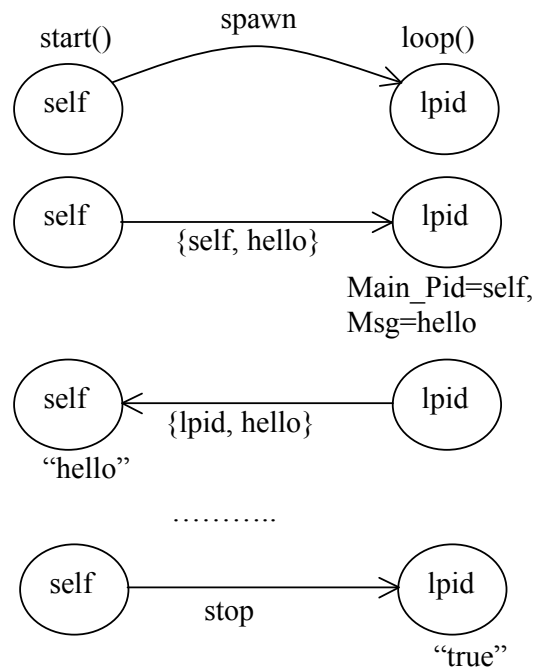


Figure 5.4 Schematic diagram of the echo process of Program 5.5

In Figure 5.4, it is shown that for the *spawn* function, processes start and loop are working in parallel. While they are executed in parallel, communications between them are described with the label graph(s). It is clear that start process sends a tuple of message $\{self, hello\}$, its PID *self* and an atom *hello* to the loop process (its PID is *lpid*). Upon reception of such a tuple of message from the start process, loop process sends back the same message (here *hello*) to the start process along with its process identifier *lpid*. Start process finally sends a message *stop* to the loop process.

Loop process can receive further message(s) from start process or can receive message *true* and thereby, terminates execution.

5.5.6(b) Translation in the π -calculus

According to rule(21),

main() \rightarrow new self(start(self))

TrPI_{exp}(self, start()) \rightarrow

```

    Loop_Pid = spawn(loop, []),
    Loop_Pid ! {self(), hello},
    receive
        { Loop_Pid, Msg}  $\rightarrow$  Msg
    end,
    Loop_Pid ! stop.)

```

(20A) \rightarrow (11) \rightarrow (13) \rightarrow (25) \rightarrow (26A) \rightarrow (17A) (only major rules are marked in sequence)
 \rightarrow start(self) \rightarrow new lpid, p, loop_res, start_send1_res, start_receive_res, start_send2_res
 (p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self,
 hello>.nil || start_send1_res'<self, hello>.nil || start_send1_res(dummy).(
 self(Loop_Pid, Msg).start_receive_res'<Msg>.nil || start_receive_res(dummy). (Loop_Pid'
 <stop>.nil || start_send2_res'<stop>.nil))))

TrPI_{exp}(self, loop()) \rightarrow

```

    receive
        {Main_Pid, Msg}  $\rightarrow$ 
            Main_Pid ! {self(), Msg},
            loop();
    stop  $\rightarrow$ 
        true
    end.)

```


(20A) \rightarrow (14) \rightarrow 26A) \rightarrow (13) \rightarrow (25) \rightarrow (15) (only major rules are marked in sequence, **loop_res** is restricted to the whole system)

```
=loop(self)=self(Main_Pid, Msg).( Main_Pid'<self, Msg>.nil || loop_send_res'<self,
Msg>.nil || loop_send_res(dummy). loop(self) ) + self(input_pat).[input_pat=stop]
loop_res'<true>.nil
```

5.5.6(c) The π -Model

From Section 5.5.6(b), the π -model of Program 5.5 can be written as follows:

```
main()=new self(start(self))
```

```
start(self)=new lpid, p, loop_res, start_send1_res, start_receive_res, start_send2_res
(p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil || p(Loop_Pid). ( Loop_Pid'<self,
hello>.nil || start_send1_res'<self, hello>.nil || start_send1_res(dummy).(
self(Loop_Pid, Msg).start_receive_res'<Msg>.nil || start_receive_res(dummy). (
Loop_Pid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

```
loop(self)=self(Main_Pid, Msg).( Main_Pid'<self, Msg>.nil || loop_send_res'<self,
Msg>.nil || loop_send_res(dummy). loop(self) ) + self(input_pat).[input_pat=stop]
loop_res'<true>.nil
```

5.5.6(d) Observing Behavior in π -calculus: Rule (26A) is Insufficient

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))
```

```
(substituting RHS of process start(self))
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, start_send2_res (
p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil || p(Loop_Pid). ( Loop_Pid'<self,
hello>.nil || start_send1_res'<self, hello>.nil || start_send1_res(dummy).(
self(Loop_Pid, Msg).start_receive_res'<Msg>.nil || start_receive_res(dummy). (
Loop_Pid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

```
(react on p)  $\rightarrow$ (Omitting nil process)
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, start_send2_res (
loop(lpid) || loop_res(dummy).nil || ( lpid'<self, hello>.nil || start_send1_res'<self,
hello>.nil || start_send1_res(dummy).( self(Loop_Pid, Msg).
start_receive_res'<Msg>.nil || start_receive_res(dummy). ( lpid'<stop>.nil ||
start_send2_res'<stop>.nil ) ) ) )
```

As a result of *react* rule on p , every free occurrences of $Loop_Pid$ will be replaced by $lpid$. One occurrence of $Loop_Pid$ is bound with a receive action along $self$. As this occurrence is bound with $self$, while *react* on $self$, any *name* can be received in $Loop_Pid$ along $self$ which breaks the semantics of its corresponding Erlang Program 5.5. Since $Loop_Pid$ is already bound with a value before used in *receive* expression match pattern, it cannot be bound 2nd time. But in translated π -model (with rule (26A)), $Loop_Pid$ can be further bound with any received term along channel $self$ which obviously breaks the semantics of Erlang.

We also know that in π -calculus two process expressions are structurally congruent, if one can be transformed into the other by renaming of bound names. As $Loop_Pid$ is bound in the above process expression, we can apply renaming rules on it.

(renaming of bound name ***Loop_Pid*** to ***Input***)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, start_send2_res (
loop(lpid) || loop_res(dummy).nil || ( lpid'<self, hello>.nil || start_send1_res'<self,
hello>.nil || start_send1_res(dummy).( self(Input, Msg). start_receive_res'<Msg>.nil
|| start_receive_res(dummy). ( lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

Now it is clear that we are in wrong path in translation mapping as there is no existence of this occurrence of $Loop_Pid$ any more where it is supposed to be the PID of loop process, $lpid$ in pattern matching. Thereby, an improvement of rule (26A) is required(cf. Section 5.5.7).

5.5.7 An Approach to Improve Rule (26A)

First, let us consider the first two lines of the *start()* function of Program 5.5.

```
Loop_Pid = spawn(loop, []),
Loop_Pid ! {self(), hello},
```

The spawn call of the first line of the code causes the loop process(loop function) to be started in parallel with calling function (here start process) and at the same time, process identity of the loop process is assigned to variable $Loop_Pid$ and tuple of message $\{self(), hello\}$ is sent to the loop process. It has been done by the 2nd line of the code above as PID of the loop process is assigned in variable $Loop_Pid$ and the tuple of message is sent to that process whose ID is stored in $Loop_Pid$.

A variable can be bound only once in PIErlang. A variable which value has been assigned is said to be bound, otherwise, it is said to be unbound. Once a variable has been bound its value can never be changed and we want to keep this also in π -

calculus semantics. In this Program 5.5, we see that variable *Loop_Pid* has been assigned the process identifier of the loop process and the same variable is used as pattern in the *receive* statement within the same function *start()*(lines 4 of the *start* function).

While translating the receive statement,

```

receive
    {Loop_Pid, Msg} -> Msg
end

```

using rule (26A), this variable *Loop_Pid* will be considered as an *unbound* variable. In rule (26A), we have considered that any variable as a pattern of *receive* statement will be treated as an *unbound* variable and could be matched with any received message.

In PIERlang, a bound variable is always considered as bound which implies *Loop_Pid* is bound here too since it has already assigned a value with the assignment expression *Loop_Pid = spawn(loop, [])*. The tuple of pattern *{Loop_Pid, Msg}* in the *receive* statement expects to receive a tuple of size 2 where first element of the tuple is the PID of the loop process and this PID will be used for pattern matching. This is because, variable *Loop_Pid* within this tuple pattern is already bound with the PID of the loop process. This is not the same for case of the 2nd element of the tuple pattern. Here variable *Msg* is still unbound within this start function(start process) and rule (26A) will be sufficient to handle this variable pattern.

5.5.7(a) Modification of Rule (26A): Providing Bound Variable Set

To meet the semantics of PIERlang in translated π -calculus system, we have considered that a bound variable has the same nature as an atom when it is used as a pattern in the *receive* statement. Therefore, we have changed rule(26A) with an additional information for a variable, whether it is bound or unbound. If the variable is already bound before used as pattern in the *receive* statement within the same function, we will treat that variable as an atom and will follow the translation procedure for an atom(cf. rule (26A)). If it is unbound then same treatment will be followed as in rule (26A). The modified version of rule (26A) is given below:

$$\begin{aligned}
& \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\
& \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms, Numbers, Bound Variables}\} \end{array} \right. \\
& \left\{ \begin{array}{l} \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Unbound Variables}\}, \text{ where } P_i \text{ is a} \\ \text{unbound variable } X. \end{array} \right. \quad \text{-(26B)}
\end{aligned}$$

5.5.7(b) Modification of Rule (26B): Providing Bound Name Set(BNS)

While working with rule (26B), we have faced the problem of separating bound and unbound variables. We have found that we can overcome this problem in the following two ways:

- i. We can pick out the sets of bound and unbound variables of the whole PIERlang function before start translation mapping and we can use this sets while translating a variable within this function especially when a variable is used as a pattern in the *receive* statement match. It has to be noticed that each function definition has to be dealt separately while translating. Rule (26B) is based on this concept of checking bound and unbound variables. However, it would be a tedious works to find the bound and unbound sets of variables for a large complex PIERlang function definition.
- ii. It is possible to handle this problem of bound and unbound variables during translation mapping. In that case, we will maintain a set of the bound names(cf. Chapter 2) while translation mapping. When a variable is used for translation, we have to check whether there is a *name* with the same spelling and context of the variable in the Bound Name Set(BNS). If there is already a *name* same as the variable in *BNS*, we have considered that this variable is already bound with some values and we cannot used this variable as an unbound variable in *receive* expression match. Rule (26B) has been modified to meet this concept of handing bound and unbound variables with rule(26C) as follows:

$$\begin{aligned}
& \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\
& \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNS} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \text{variable } X \text{ and name } X \notin \text{BNS} \end{array} \right. \quad \text{-(26C)}
\end{aligned}$$

Where $\text{BNS} = \{ X \mid y(\dots, X, \dots) \text{ is a receive action in } \pi\text{-calculus} \\ , z \mid (\text{new } z) \text{ in } \pi\text{-Calculus} \}$

5.5.7(c) Betterment of Rule (26C) over Rule(26B)

First, consider only the receive statement code segment:

```

start()->
    receive
        {Loop_Pid, Msg} -> Msg
    end
    .....

```

Here both pattern variables *Loop_Pid* and *Msg* are unbound variables. Rule (26B) can be used here as we know in advanced that pattern variables are unbound. But consider the case of very large program where each function could be also very large. In the case of large code within a function, finding the bound and unbound variables would be tedious and time consuming. To overcome this problem, we have considered rule (26C). With rule(26C), whenever a variable is found as pattern of the *receive* statement during translation mapping, the translator will search in *BNS* whether with the same spelling as the variable, there is a name or not. While translating a complete PIERlang program, we will consider separate *BNSs* for each of the function definition to keep both the semantics of PIERlang and π -calculus. Thus, while using rule(26C) for the above code segment, it will directly say that both the pattern variables *Loop_Pid* and *Msg* are not elements of *BNS* since initially *BNS* is empty.

Using rule (26C),

```

TrPIexp(self,
    receive
        {Loop_Pid, Msg} -> Msg
    end )
(26C)
= self(Loop_Pid, Msg).TrPIexp(self, Msg)

```

The translation result implies that if any message of tuple size 2 is sent along channel *self*, the corresponding message elements will be received in names *Loop_Pid* and *Msg*.

5.5.8 PIERlang Program 5.5: Rule (26C) is Sound but has Extra Names in BNS

In this section rule (26C) is used for translating Program 5.5 and found that this rule is sound enough to translate tuple-based Match of *receive* expression. It is also noticed that the BNS set of this rule contains some extra names(cf. Section 5.5.9).

5.5.8(a) Execution in PIERlang Compiler

Execution of Program 5.5 in PIERlang compiler can be found in Section 5.5.6(a).

5.5.8(b) Translation in the π -calculus

According to rule(21),
`main()=new self(start(self))`

Using rule (26C), the start function of Program 5.5 can be translated as follows:

```
TrPIfunder(self, start() ->
    Loop_Pid = spawn(loop, []),
    Loop_Pid ! {self(), hello},
    receive
        { Loop_Pid, Msg} -> Msg
    end,
    Loop_Pid ! stop.)
```

Initially, **BNS** is empty i.e. **BNS**=`{ }`

```
(20A) -> (11)->(10A)
=start(self)=new lpid, p, loop_res ( p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
    p(Loop_Pid). ( TrPIexp( self,
        Loop_Pid ! {self(), hello},
        receive
            { Loop_Pid, Msg} -> Msg
        end,
        Loop_Pid ! stop.) ) )
```

Now **BNS**=`{lpid, p, loop_res, dummy, Loop_Pid}`

```

(13) -> (25) -> (5) (22) (3)
=new lpid, p, loop_res, start_send1_res ( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
    start_send1_res'<self, hello>.nil || start_send1_res(dummy).
    ( TrPIexp(self,
        receive
            { Loop_Pid, Msg } -> Msg
        end,
        Loop_Pid ! stop.) ) ) )

```

Now **BNS**={lpid, p, loop_res, dummy, Loop_Pid, start_send1_res}

```

(13)
=new lpid, p, loop_res, start_send1_res, start_receive_res ( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
    start_send1_res'<self, hello>.nil || start_send1_res(dummy).
    ( TrPIexp(self,
        receive
            { Loop_Pid, Msg } -> Msg
        end)
    || start_receive_res(dummy). ( TrPIexp(self, Loop_Pid ! stop.) ) ) )

```

Now **BNS**={lpid, p, loop_res, dummy, Loop_Pid, start_send1_res, start_receive_res}

Rule(26C) has to be applied now for translating the *receive* statement above. There are two variables *Loop_Pid* and *Msg* within the tuple of pattern in the *receive* statement match. According to rule (26C), we will first consider that variables *Loop_Pid* and *Msg* are now names in the π -calculus. After that we will check whether *Loop_Pid* and *Msg* are elements of *BNS* or not. It is found that *Loop_Pid* \in *BNS* but the second name *Msg* \notin *BNS*. As *Loop_Pid* \in *BNS*, we will now consider *Loop_Pid* as an atom and apply name matching feature for *Loop_Pid* like an atom. As *Msg* \notin *BNS*, we will consider *Msg* as an unbound variable in PIERlang and translate accordingly.

```

(26C)
=new lpid, p, loop_res, start_send1_res, start_receive_res ( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
    start_send1_res'<self, hello>.nil || start_send1_res(dummy).
    ( self(input_pat1, Msg ).[input_pat1=Loop_Pid] TrPIexp(self, Msg)
    || start_receive_res(dummy). ( TrPIexp(self, Loop_Pid ! stop.) ) ) )

```

BNS={lpid, p, loop_res, dummy, Loop_Pid, start_send1_res, start_receive_res,
input_pat1, Msg}

This is the core point we are dealing with. We see that we have applied name matching feature for *Loop_Pid* considering it as an atom.

(6) (9) -> (5) (3)
 =new lpid, p, loop_res, start_send1_res, start_receive_res (p'<lpid>.nil || loop(lpid) ||
 loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
 start_send1_res'<self, hello>.nil || start_send1_res(dummy).
 (self(input_pat1, Msg).[input_pat1=Loop_Pid] start_receive_res'<Msg>.nil
 || start_receive_res(dummy).(Loop_Pid'<stop>.nil ||
 start_send2_res'<stop>.nil))))

Now **BNS**={lpid, p, loop_res, dummy, Loop_Pid, start_send1_res, start_receive_res,
input_pat1, Msg}

Similarly, for the loop process of Program 5.5:

TrPI_{funder}(self, loop() ->
 receive
 {Main_Pid, Msg} ->
 Main_Pid ! {self(), Msg},
 loop();
 stop -> true
 end.)

Initially, **BNS** is empty i.e. **BNS**={ }

(20A) -> (14)
 =loop(self)=TrPI_{exp}(self,
 receive
 {Main_Pid, Msg} ->
 Main_Pid ! {self(), Msg},
 loop();
 end.) +
 TrPI_{exp}(self,
 receive
 stop -> true
 end.)
BNS={ }

(26C, as both *Main_Pid* and *Msg* \notin *BNS*)

```
=self(Main_Pid, Msg).( TrPIexp(self, Main_Pid ! {self(), Msg}, loop() ) ) +
  self(input_pat1).[input_pat1=stop] TrPIexp(self, true)
```

BNS={Main_Pid, Msg, input_pat1}

(13)->(25)->(5) (22) (10A) (4) (*loop_res* is restricted to the whole system for returning result of loop function)

```
=new loop_send_res( self(Main_Pid, Msg).( Main_Pid'<self, Msg>.nil ||
  loop_send_res'<self, Msg>.nil || loop_send_res(dummy).loop(self) ) +
  self(input_pat1).[input_pat1=stop] loop_res' <true>.nil)
```

BNS={Main_Pid, Msg, input_pat1, loop_send_res, loop_res}

Variable *Msg* is used as the patterns of both the *receive* statements of start and loop functions. According to π Erlang, it does not matter whether it is the same variable name or different as long as it is used in different function definitions. We have also kept this semantics in translated system by considering separate *BNS* for each of the function definitions, thus, there will be no problem if there are same names in different *BNS* while translating.

5.5.8(c) The π -Model

From Section 5.5.8(b), the π -model of Program 5.5 with rule(26C) can be written as follows:

```
main()=new self(start(self))
```

```
start(self)=new lpid, p, loop_res, start_send1_res, start_receive_res ( p'<lpid>.nil ||
  loop(lpid) || loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
  start_send1_res'<self, hello>.nil || start_send1_res(dummy).
  (self(input_pat1, Msg ).[input_pat1=Loop_Pid] start_receive_res'<Msg>.nil
  || start_receive_res(dummy).( Loop_Pid'<stop>.nil ||
  start_send2_res'<stop>.nil ) ) ) )
```

```
loop(self)= new loop_send_res( self(Main_Pid, Msg).( Main_Pid'<self, Msg>.nil ||
  loop_send_res'<self, Msg>.nil || loop_send_res(dummy).loop(self) ) +
  self(input_pat1).[input_pat1=stop] loop_res' <true>.nil)
```

5.5.8(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

main() = new self(start(self))

(substituting RHS of process *start(self)*)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res ( p'<lpid>.nil ||
  loop(lpid) || loop_res(dummy).nil || p(Loop_Pid). (Loop_Pid'<self, hello>.nil ||
  start_send1_res'<self, hello>.nil || start_send1_res(dummy).
  ( self(input_pat1, Msg ).[input_pat1=Loop_Pid] start_receive_res'<Msg>.nil ||
  start_receive_res(dummy).( Loop_Pid'<stop>.nil || start_send2_res'<stop>.nil ))))
```

(react on *p*, every free occurrence *Loop_Pid* is bound with PID of loop process, *lpid*) -> (Omitting nil process)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res (
  // loop process
  loop(lpid) || loop_res(dummy).nil
  || // start process
  (lpid'<self, hello>.nil || start_send1_res'<self, hello>.nil ||
  start_send1_res(dummy).( self(input_pat1, Msg ).[input_pat1=lpid]
  start_receive_res'<Msg>.nil || start_receive_res(dummy).( lpid'<stop>.nil ||
  start_send2_res'<stop>.nil ) ) ) )
```

All the occurrences of *Loop_Pid* are now free occurrence and hence are replaced by *lpid* due to applying the reaction rule on *p*. In Section 5.5.6(d), one occurrence of *Loop_Pid* has been found as bound while translating with rule (26A) and thus, there was a problem with semantic matching between π -calculus and PIERlang.

(substituting RHS of process *loop(self)* with *lpid* as parameter i.e. *self* -> *lpid*)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  // loop process
  ( lpid(Main_Pid, Msg).( Main_Pid'<lpid, Msg>.nil || loop_send_res'<lpid,
  Msg>.nil || loop_send_res(dummy).loop(lpid) ) + lpid(input_pat1).
  [input_pat1=stop] loop_res'<true>.nil ) || loop_res(dummy).nil
  || // start process
  ( lpid'<self, hello>.nil || start_send1_res'<self, hello>.nil ||
  start_send1_res(dummy).( self(input_pat1, Msg).[input_pat1=lpid]
  start_receive_res'<Msg>.nil || start_receive_res(dummy).( lpid'<stop>.nil ||
  start_send2_res'<stop>.nil ) ) ) )
```

From the PIERlang Program 5.5 and/or from its corresponding schematic diagram(Figure 5.4), we know that start process and loop process are executed in parallel. While executing in parallel, start process has the option to send a tuple of message $\{self, hello\}$ to loop process(Figure 5.4) and then it can receive a tuple of message and finally it can send an atom $stop$ to loop process in sequence. We see here that the translated π -calculus system shows the same behavior. The start process and loop process work in parallel and tuple of message $\{self, hello\}$ is sent along channel $lpid$. We know $lpid$ is the PID of loop process and hence its clear that this tuple of message is sent to the loop process. Although loop process is ready to receive a tuple of message (size 2) or a single atom, it can only receive the tuple of message since the start process only sends the tuple of message first. Start process will then receive a tuple of message and finally, will send the message $stop$. We have found the same behavior in translated system. Start process is ready to send tuple of message with $lpid' <self, hello>.\underline{nil}$ and atom of message with $start_receive_res(dummy).(lpid' <stop>.\underline{nil} \parallel start_send2_res' <stop>.\underline{nil})$. We see that before sending atom $stop$ to loop process there must be a dummy communication along the receiver channel ($start_receive_res$) of start process which means that receiving of tuple of start process has to be performed first then then it can send message $stop$ to loop process. In this way, we keep the sequene of execution of expressions in translated system, the same way we can expect from PIERlang program.

```
(react on lpid, loop process has received a tuple of message from start process)
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (      // loop process
    self'<lpid, hello>.nil || loop_send_res'<lpid, hello>.nil ||
    loop_send_res(dummy).loop(lpid) ) || loop_res(dummy).nil
    || // start process
    (nil || start_send1_res'<self, hello>.nil || start_send1_res(dummy).(
      self(input_pat1, Msg).[input_pat1=lpid] start_receive_res'<Msg>.nil ||
      start_receive_res(dummy).( lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

Now, loop process sends a tuple of message (its PID $lpid$ and atom $hello$) to start process (PID $self$). We have found that the same behavior from PIERlang Program 5.5 and also from its corresponding schematic diagram (Figure 5.4). But start process has to perform a dummy communication over its first sender channel named $start_send1_res$. This dummy communication forces the start process to maintain the sequence of execution of expressions as we have mentioned above. Here a communication along channel $start_send1_res$ means that start process has to send the tuple of message before receiving since sending expression is ahead of receiving

expression in program 5.5. However, to have a communication over *start_send1_res*, we are required to adapt the receiver side channel with sender one with *CAR for DC*(cf. Section 5.5.4(d)).

(*CAR for DC* on receiver side channel *start_send1_res*)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (      // loop process
    self'<lpid, hello>.nil || loop_send_res'<lpid, hello>.nil ||
    loop_send_res(dummy).loop(lpid) ) || loop_res(dummy).nil
    || // start process
    (nil || start_send1_res'<self, hello>.nil || start_send1_res(dummy1, dummy2).(
      self(input_pat1, Msg).[input_pat1=lpid] start_receive_res'<Msg>.nil ||
      start_receive_res(dummy).( lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

(react on *start_send1_res*)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (      // loop process
    self'<lpid, hello>.nil || loop_send_res'<lpid, hello>.nil ||
    loop_send_res(dummy).loop(lpid) ) || loop_res(dummy).nil
    || // start process
    (nil || nil || ( self(input_pat1, Msg).[input_pat1=lpid]
      start_receive_res'<Msg>.nil || start_receive_res(dummy).( lpid'<stop>.nil ||
      start_send2_res'<stop>.nil ) ) ) )
```

(react on *self*, now its possible for the start process to receive message *{lpid, hello}* from loop process)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (      // loop process
    nil || loop_send_res'<lpid, hello>.nil ||
    loop_send_res(dummy).loop(lpid) ) || loop_res(dummy).nil
    || // start process
    (nil || nil || ( [lpid=lpid] start_receive_res'<hello>.nil
    || start_receive_res(dummy).( lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

A successful name matching [*lpid=lpid*] is found which implies that message *hello* is now sent along receiver channel (*start_receive_res*) of the start process. Now start process can send message *stop* to the loop process but before doing so there should be a dummy communication over channel *start_receive_res*, for the same reason we have mentioned before, to maintain the sequence of execution of expressions.

(react *start_receive_res*, now its possible for the start process to send message *stop* to loop process)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (
    // loop process
    nil || loop_send_res'<lpid, hello>.nil ||
    loop_send_res(dummy).loop(lpid) || loop_res(dummy).nil
    || // start process
    (nil || nil || (nil || (lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

We know from Program 5.5 and/or from Figure 5.4 that loop process can receive a tuple of message and then starts the loop process again or it can receive the message *stop*. If it receives atom *stop*, further execution will be terminated by evaluating atom *true*. As loop process has received a tuple of message, it can now start again to receive further messages (tuple of message or message *stop*). However, before restarting the loop process, there should a dummy communication over channel *loop_send_res*, which binds the loop process to be started again only after performing the send operation ahead of it. We also notice that we are required to adapt (CAR for DC) the receiver side of channel *loop_send_res* to have a communication.

(CAR for DC on receiver side channel *loop_send_res*) -> (react on *loop_send_res*)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (
    // start process
    nil || nil || loop(lpid) || loop_res(dummy).nil
    || // start process
    (nil || nil || (nil || (lpid'<stop>.nil || start_send2_res'<stop>.nil ) ) ) )
```

(Omitting the inactive *nil* processes)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (
    // loop process
    loop(lpid) || loop_res(dummy).nil
    || // start process
    (lpid'<stop>.nil || start_send2_res'<stop>.nil) )
```

(substituting the RHS of loop process with *lpid* as parameter i.e. *self->lpid* , in this way it can now be restarted again)

```
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
  (
    // loop process
    lpid(Main_Pid, Msg).( Main_Pid'<lpid, Msg>.nil ||
    loop_send_res'<lpid, Msg>.nil || loop_send_res(dummy).loop(lpid) ) +
    lpid(input_pat1).[input_pat1=stop] loop_res' <true>.nil )
```

```

    || // start process
    ( lpid'<stop>.nil || start_send2_res'<stop>.nil) )

```

Start process sends a message *stop* to loop process and loop process is ready to receive that atom too.

```

(react on lpid)
=>new self, lpid, p, loop_res, start_send1_res, start_receive_res, loop_send_res (
    ( // loop process
      [stop=stop] loop_res' <true>.nil )

      || // start process
      ( nil || start_send2_res'<stop>.nil) )

```

A successful name matching [*stop*=*stop*] is found and consequently, atom *true* is sent over the result channel of loop process and thus, terminates further execution and communication with start process. As we see, π -model now works well meeting the semantics of PIERlang therefore, we can say that rule(26C) works correctly. However, its BNS contains some extra names(cf. Section 5.5.9).

5.5.9 Improving Rule (26C): Providing BNS with Receive Actions of Variables

While working with rule (26C), we have observed that it is not required to work with the bound names those are included to *BNS* due to the *new* operator of π -calculus. This is because, we are dealing with bounding of a variable in PIERlang and this can only be done with a *receive action* in π -calculus. We have also noticed that we are not required to consider all the bound names of the *receive action*. We only need to consider those bound names of *receive action* that are *variables* in PIERlang program. Thereby, we have modified the definition of *BNS* and renamed *Bound Name Set* (*BNS*) as *Bound Names Set with Receive Actions of Variables* (*BNSRAV*).

We can now redefine rule (26C) to rule (26D) with the modified definition of *BNS*. Rule(26D) will work for both monadic and polyadic communications. If ($n=1$), rule (26D) will be capable of handling atomic arguments as of rules (15), (16) and (17A). Thereby, from now rule (26D) will be used for any kind (tuple-based & atomic arguments) of pattern matching in Matches of *receive* and *case* expressions

$$\text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \begin{cases} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNSRAV} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \text{variable } X \text{ and name } X \notin \text{BNSRAV} \end{cases} \quad \text{-(26D)}$$

Where $\text{BNSRAV} = \{ X \mid y(\dots, X, \dots) \text{ is a receive action in } \pi\text{-calculus and } X \text{ is a variable in PIERlang function for which } X \text{ is considered as a name in } \pi\text{-calculus.} \}$

5.5.10 PIERlang Program 5.6: Rule (26D) Sounds Perfect

We will now consider another example, the so called locker process [2] to verify rule(26D) where a variable is bound with pattern matching in *receive* statement rather than assignment in the case of echo process in Program 5.5(cf. Section 5.5.8).

```
start() ->
    Locker_Pid = spawn(locker, []),
    spawn(client, [Locker_Pid]),
    spawn(client, [Locker_Pid]).

locker() ->
    receive
        {request, Client} ->                // Client is bound here
            Client !ok,
            receive
                {release, Client} ->        // Client Behaves like atom
                    locker()
            end
        end
    end.

client(Locker_Pid) ->
    Locker_Pid ! {request, self()},
    receive
        ok -> do_critical_works,
            Locker_Pid ! {release, self()},
            client(Locker_Pid)
    end.
```

Program 5.6 A simple locker process.

5.5.10(a) Execution in PIErlang Compiler

The execution behavior of Program 5.6 is shown in Figure 5.5. More details can be found in Section 5.5.10(d) while observing its π -model behavior in π -calculus.

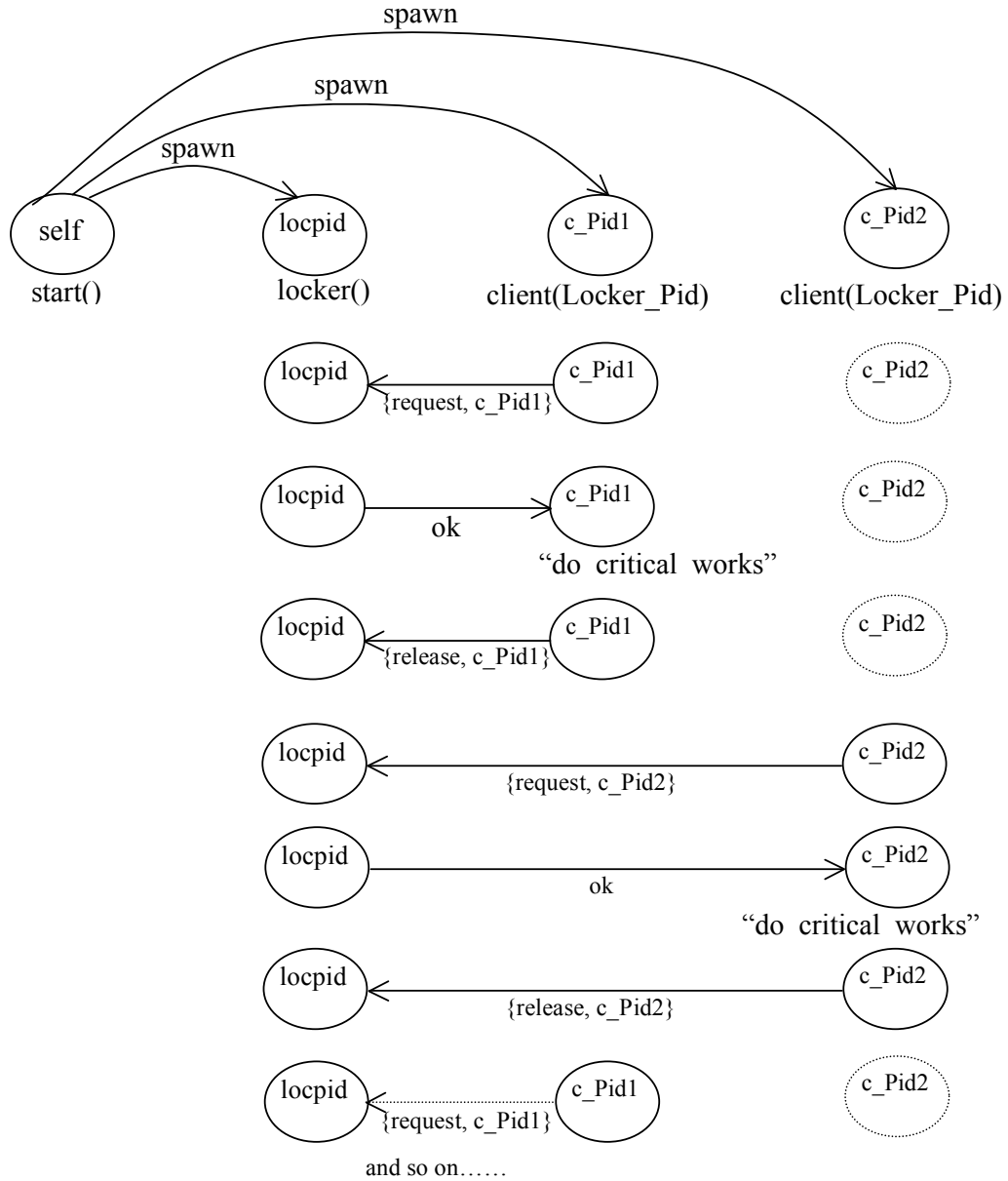


Figure 5.5 Communicating schematic diagram of locker process Program 5.6.

5.5.10(b) Translation in the π -calculus

In function `locker()` of Program 5.6, variable *Client* is used twice within a nested *receive* statement. According to Erlang semantics, *Client* is bound with a value by the first *receive* statement and then in the second nested *receive* statement it behaves like

atom. The translation procedure of Program 5.6 is described below with rule (26D) by considering such kind of variable binding:

According to rule(21),
main()=new self(start(self))

```
TrPIfunder(start() ->
    Locker_Pid=spawn(locker, []),
    spawn(client, [Locker_Pid]) ,
    spawn(client, [Locker_Pid]). )
```

Initially, **BNSRAV** is empty i.e. **BNSRAV**= $\{ \}$

[illegible]
$$\mathbf{BNSRAV} = \{ \}$$

```
(11, we suppose locpid is the PID of the locker process) ->(10A)
=new locpid, p, locker_res ( p' < locpid > .nil || locker(locpid) || locker_res(dummy).nil ||
    p(Locker_Pid). (TrPIexp(self, spawn(client, [Locker_Pid]), spawn(client,
    [Locker_Pid] ) ) ) ) )
```

$$\mathbf{BNSRAV} = \{\text{Locker_Pid}\}$$

There are two spawn calls for the same client function definition. We suppose that the first and second spawn calls to client process will return the client process identifiers *c_Pid1* and *c_Pid2* respectively.

```
(12) ->(10)->(12)->(10)
=new locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res (
    p'<locpid>.nil || locker(locpid) || locker_res(dummy).nil || p(Locker_Pid).
    client(c_Pid1, Locker_Pid) || client1_res(dummy).nil || client(c_Pid2,
    Locker_Pid) || client2_res(dummy).nil ))
```

$$\mathbf{BNSRAV} = \{\text{Locker Pid}\}$$

Now for the locker process:

```

TrPIfunder(self, locker() ->
    receive
    {request, Client} ->
        Client !ok,
    receive
        {release, Client} ->
            locker()
    end
end.)

```

Initially, **BNSRAV** is empty i.e. **BNSRAV**={ }

(20A) ->(14)->(26D, Client \notin **BNSRAV**, so its still unbound variable)
 =locker(self)= self(input_pat1, **Client**).[input_pat1=request] (TrPI_{exp}(self, Client !ok,
 receive
 {release, Client} ->
 locker()
 end))

Now **BNSRAV**={Client} ; variable Client is used in *receive action* **self(input_pat1, Client)**

(13) ->(9) -> (5) (3) (14)->(26D, Client \in **BNSRAV**, so now **Client** will be considered as an atom)->(10A)
 =new locker_send_res (self(input_pat1, Client).[input_pat1=request] (Client'<ok>.nil
 || locker_send_res'<ok>.nil || locker_send_res(dummy). (self(input_pat2,
 input_pat3).[input_pat2=release][**input_pat3=Client**] locker(self))))

Now **BNSRAV**={Client}

Now for the client process:

```

TrPIfunder(self, client(Locker_Pid) ->
    Locker_Pid !{request, self()},
    receive
    ok ->do_critical_works,
    Locker_Pid !{release,self()},
    client(Locker_Pid)
end.)

```

Initially, **BNSRAV** is empty i.e. **BNSRAV**={ }

```

(20)->(13) ->(25) ->(5) (3) (22) ->(14)
=client(self, Locker_Pid)=new client_send1_res ( Locker_Pid'<request, self>.nil ||
  client_send1_res'<request, self>.nil || client_send1_res(dummy). (
    TrPImatch(self, ok ->do_critical_works,
      Locker_Pid !{release,self()}},
    client(Locker_Pid) ) ) )

```

BNSRAV={ }

```

(26D)->(13)->(4) ->(25)->(5) (3) (22)->(13)->(10)->(5)
=new client_send1_res, client_exp1_res, client_exp2_res ( Locker_Pid'<request,
  self>.nil || client_send1_res'<request, self>.nil || client_send1_res(dummy).(
    self(input_pat1).[input_pat1=ok] ( client_exp1_res'<do_critical_works>.nil ||
    client_exp1_res(dummy).( Locker_Pid'<release, self>.nil ||
    client_exp2_res'<release, self>.nil || client_exp2_res(dummy). client(self,
    Locker_Pid) ) ) ) )

```

BNSRAV={ }

5.5.10(c) The π -Model

From Section 5.5.10(b), the π -model of Program 5.6 can be written as follows:

```
main()=new self(start(self))
```

```

start(self)= new locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res (
  p'<locpid>.nil || locker(locpid) || locker_res(dummy).nil || p(Locker_Pid). (
    client(c_Pid1, Locker_Pid) || client1_res(dummy).nil || client(c_Pid2,
    Locker_Pid) || client2_res(dummy).nil ) )

```

```

locker(self)= new locker_send_res ( self(input_pat1, Client).[input_pat1=request]
  (Client'<ok>.nil || locker_send_res'<ok>.nil || locker_send_res(dummy). (
    self(input_pat2, input_pat3).[input_pat2=release][input_pat3=Client]
    locker(self) ) ) )

```

```

client(self, Locker_Pid)= new client_send1_res, client_exp1_res, client_exp2_res (
  Locker_Pid'<request, self>.nil || client_send1_res'<request, self>.nil ||
  client_send1_res(dummy).(
    self(input_pat1).[input_pat1=ok] ( client_exp1_res'<do_critical_works>.nil ||
    client_exp1_res(dummy).( Locker_Pid'<release, self>.nil ||
    client_exp2_res'<release, self>.nil || client_exp2_res(dummy). client(self,
    Locker_Pid) ) ) ) )

```

5.5.10(d) Observing Behavior in π -calculus:

To observe the model behavior in π -calculus, we have to start from the *main()* process.

`main()=new self(start(self))`

(substituting RHS of process *start(self)*)

`=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res (`
`p'<locpid>.nil || locker(locpid) || locker_res(dummy).nil || p(Locker_Pid). (`
`client(c_Pid1, Locker_Pid) || client1_res(dummy).nil || client(c_Pid2, Locker_Pid) ||`
`client2_res(dummy).nil))`

Now we can apply the reaction rule on *p*, thus all free occurrence of *Locker_Pid* will be replaced by *locpid*, PID of locker process. All the occurrences of *Locker_Pid* are free and hence will be replaced by *locpid*. Here *client(c_Pid1, Locker_Pid)* and *client(c_Pid2, Locker_Pid)* are two process calls to the client process with different PIDs.

(react on *p*) ->(Omitting *nil* process)

`*=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res (`
`// locker process`
`locker(locpid) || locker_res(dummy).nil`
`|| // first instance of client process`
`client(c_Pid1, locpid) || client1_res(dummy).nil`
`|| // second instance of client process`
`client(c_Pid2, locpid) || client2_res(dummy).nil)`

Here in the above π -model, locker process and two instances of the client processes are started running in parallel. The same thing is found in PIERlang Program 5.6 and its corresponding communication schematic diagram in Figure 5.5. At that moment, either the first or the second instance of the client process can start communication with the locker process by sending a request to the locker process. From Figure 5.5, we see that first instance of the client process sends a request along with its PID to the locker process.

After receiving the request from first client instance, locker process allows it to do critical works by sending an *ok* message. After finishing the critical works, it(first client instance) send a release message to the locker process, thus releasing the locker. At that moment, a new instance (either first or second instance of client process) of the client process can send request to the locker process to get the locker for doing

critical works. In Figure 5.5, we see that the second instance of the client process sends the request to the locker process. After getting request from client process (second instance), locker process grants it to do critical works. After finishing critical works, client process (second instance) sends a release message to the locker process, thus releasing the locker and so on.

(substituting RHS definitions of locker process with *self->locpid*, first client process with *self->c_Pid1*, *Locker_Pid ->locpid*)

```
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(

  // locker process
  ( locpid(input_pat1, Client).[input_pat1=request] (Client'<ok>.nil ||
locker_send_res'<ok>.nil || locker_send_res(dummy). ( locpid(input_pat2,
input_pat3).[input_pat2=release][input_pat3=Client] locker(locpid) ) ) ||
locker_res(dummy).nil

|| // first instance of client process

  locpid'<request, c_Pid1>.nil || client_send1_res'<request, c_Pid1>.nil ||
client_send1_res(dummy).( c_Pid1(input_pat1).[input_pat1=ok] (
client_exp1_res'<do_critical_works>.nil || client_exp1_res(dummy).(
locpid'<release, c_Pid1>.nil || client_exp2_res'<release, c_Pid1>.nil ||
client_exp2_res(dummy). client(c_Pid1, locpid) ) ) ) || client1_res(dummy).nil

|| // second instance of client process
client(c_Pid2, locpid) || client2_res(dummy).nil )
```

According to Figure 5.5, we also suppose that first instance (PID *c_Pid1*) of the client process requests the locker process to get permission to do critical works.

```
(react on locpid)
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(

  // locker process
  ( [request=request] (c_Pid1'<ok>.nil || locker_send_res'<ok>.nil ||
locker_send_res(dummy). ( locpid(input_pat2,
input_pat3).[input_pat2=release][input_pat3=c_Pid1] locker(locpid) ) ) ||
locker_res(dummy).nil
```

```

|| // first instance of client process

nil || client_send1_res'<request, c_Pid1>.nil || client_send1_res(dummy).(
c_Pid1(input_pat1).[input_pat1=ok] ( client_exp1_res'<do_critical_works>.nil ||
client_exp1_res(dummy).( locpid '<release, c_Pid1>.nil || client_exp2_res'<release,
c_Pid1>.nil || client_exp2_res(dummy). client(c_Pid1, locpid) ) ) ) ||
client1_res(dummy).nil

|| // second instance of client process
client(c_Pid2, locpid) || client2_res(dummy).nil
)

```

In locker process, name matching [*request*=request] is found, consequently, it sends permission message *ok* to the first client instance (PID *c_Pid1*) but client process is not ready to receive this. To do so, it has to perform a dummy communication on channel *client_send1_res*.

(CAR for DC on **client_send1_res**)-> (react on **client_send1_res**)->(name matching [**request**=request] is found) -> (react on **c_Pid1**)
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(

```

// locker process
((nil || locker_send_res'<ok>.nil || locker_send_res(dummy). ( locpid(input_pat2,
input_pat3).[input_pat2=release][input_pat3= c_Pid1] locker(locpid) ) ) ) ||
locker_res(dummy).nil

```

```

|| // first instance of client process

nil || nil || ( [ok=ok] ( client_exp1_res'<do_critical_works>.nil ||
client_exp1_res(dummy).( locpid '<release, c_Pid1>.nil || client_exp2_res'<release,
c_Pid1>.nil || client_exp2_res(dummy). client(c_Pid1, locpid) ) ) ) ||
client1_res(dummy).nil

|| // second instance of client process
client(c_Pid2, locpid) || client2_res(dummy).nil
)

```

In the first client process, a name matching [*ok*=ok] is found and thus, it can now do critical works.

```

(name matching [ok=ok] is found)->(first client instance is now can do critical works
) -> (react on client_exp1_res)
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(

    // locker process
    ((nil || locker_send_res'<ok>.nil || locker_send_res(dummy).( locpid(input_pat2,
input_pat3).[input_pat2=release][input_pat3= c_Pid1] locker(locpid) ) ) ) ||
locker_res(dummy).nil

|| // first instance of client process

nil || nil || ( (nil || ( locpid '<release, c_Pid1>.nil || client_exp2_res'<release,
c_Pid1>.nil || client_exp2_res(dummy). client(c_Pid1, locpid) ) ) ) ||
client1_res(dummy).nil

|| // second instance of client process
client(c_Pid2, locpid) || client2_res(dummy).nil
)

```

Client process (First instance) sends the release message to the locker process. But before receiving, locker has to perform a dummy communication to maintain the sequence of execution of expressions on channel *locker_send_res*.

```

(react on locker_send_res) ->(react on locpid)
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(

    // locker process
    ((nil || nil || ( [release=release][c_Pid1= c_Pid1] locker(locpid) ) ) ) ||
locker_res(dummy).nil

|| // first instance of client process

nil || nil || ( (nil || (nil || client_exp2_res'<release, c_Pid1>.nil ||
client_exp2_res(dummy). client(c_Pid1, locpid) ) ) ) || client1_res(dummy).nil

|| // second instance of client process
client(c_Pid2, locpid) || client2_res(dummy).nil
)

```

A successful name matching [*release=release*][*c_Pid1*= c_Pid1] is found and consequently locker process can now start again for serving next client request.

(CAR for DC on *client_exp2_res*) -> (react on *client_exp2_res*, for maintaining sequence of execution of expressions)

```
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(
```

```
    // locker process
```

```
((nil || nil || (locker(locpid) ) ) ) || locker_res(dummy).nil
```

```
|| // first instance of client process
```

```
nil || nil || ( (nil || (nil || nil || client(c_Pid1, locpid) ) ) ) || client1_res(dummy).nil
```

```
|| // second instance of client process
```

```
client(c_Pid2, locpid) || client2_res(dummy).nil
```

```
)
```

(Omitting *inactive nil* processes)

```
=>new self, locpid, p, locker_res, c_Pid1, c_Pid2, client1_res, client2_res,
locker_send_res, client_send1_res, client_exp1_res, client_exp2_res(
```

```
    // locker process
```

```
locker(locpid) || locker_res(dummy).nil
```

```
|| // first instance of client process
```

```
client(c_Pid1, locpid) || client1_res(dummy).nil
```

```
|| // second instance of client process
```

```
client(c_Pid2, locpid) || client2_res(dummy).nil
```

```
)
```

Now we see that we are now again on the same initial state(* above) where locker process and two instances client processes work in parallel. At that moment, again first instance of client process can send a request to the locker process for granting permission of doing critical works or the second instance of the client process has the same chance to request the locker process for getting permission of doing critical works. In Figure 5.5, we see that second instance of the client process sends request to the locker process for getting permission of doing critical works. As per the request, locker process grants permission to the second client process and thus, allowing it to do critical works and finally, it (second instance of client process) sends release information to the locker process and so on.

The main reason behind using this locker process is to verify the soundness of rule (26D). We see that translated π -model with rule (26D) works accurately in π -calculus

and shows the same behavior like its corresponding PIERlang program, thereby, rule (26D) sounds perfect.

5.6 Tuples in Case Expressions

The basic concept of working with tuples in *case* expression is the same as already discussed in Section 4.8.2 with rule (19). The only modification to this rule is the adaptation of *case_res* channel from monadic to polyadic form for supporting tuples in communication. To deal with the tuple-based matches in *case* expression, rule (26D) will be used. For evaluating the tuple-based *case* head, rule (24) will be used with *res* replaced by *case_res*. As in Section 4.8, all variables used in *case head* must be bound.

5.6.1 PIERlang Program 5.7: Tuples in Case Expression

Let us consider the following Program 5.7 where *case* expression is used:

```
start()-> Weekday=monday,
          Work=swimming,
          spawn(loop, [Weekday, Work]).

loop(X, Y)->
    case {X, hello, Y} of
        {monday, hello, swimming} -> go_for_swim;
        {Weekend, hi, rest} -> take_rest;
        Undefined -> sleep
    end.
```

Program 5.7: A simple case-based message passing process.

5.6.1(a) Execution in PIERlang Compiler

In Figure 5.6, it is shown that for the *spawn* function, processes start and loop work in parallel. While they work in parallel, communications between them are described with the label graph.

The loop function within spawn call(in start function) is initiated with its arguments *Weekday* and *Work*. They already assigned values *monday* and *swimming* respectively. When loop function is called variables X and Y in loop function are bound with values *monday* and *swimming* respectively. In loop process there is a case

expression which uses these values and for a successful pattern matching *go_for_swim* is returned as the result of the *case* expression and thus, result of the whole loop process.

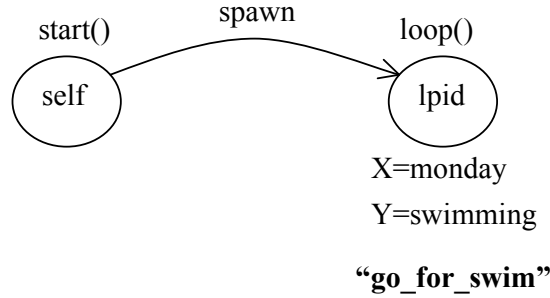


Figure 5.6 Schematic diagram of the Program 5.7

5.6.1(b) Translation in the π -calculus

According to rule(21),

$\text{main}() = \text{new self}(\text{start}(\text{self}))$

$\text{TrPI}_{\text{exp}}(\text{self}, \text{start}() \rightarrow \text{Weekday} = \text{monday},$
 $\text{Work} = \text{swimming},$
 $\text{spawn}(\text{loop}, [\text{Weekday}, \text{Work}])).$

Initially, **BNSRAV** is empty i.e. **BNSRAV** = { }

(20A) $(7) \rightarrow (7) \rightarrow (12) \rightarrow (10)(5)$
 $= \text{start}(\text{self}) = \text{new exp1_res}, \text{exp2_res}, \text{lpid}, \text{loop_res} (\text{exp1_res}' < \text{monday} > . \underline{\text{nil}} \parallel$
 $\text{exp1_res}(\text{Weekday}). (\text{exp2_res}' < \text{swimming} > . \underline{\text{nil}} \parallel \text{exp2_res}(\text{Work}). (\text{loop}(\text{lpid},$
 $\text{Weekday}, \text{Work}) \parallel \text{loop_res}(\text{dummy}))))$

Finally, **BNSRAV** = { Weekday, Work }

$\text{TrPI}_{\text{exp}}(\text{self}, \text{loop}(X, Y) \rightarrow$
 $\text{case } \{X, \text{hello}, Y\} \text{ of}$
 $\{ \text{monday}, \text{hello}, \text{swimming} \} \rightarrow \text{go_for_swim};$
 $\{ \text{Weekend}, \text{hi}, \text{rest} \} \rightarrow \text{take_rest};$
 $\text{Undefined} \rightarrow \text{sleep}$
 $\text{end.})$

Initially, **BNSRAV** is empty i.e. **BNSRAV** = { }

```

(20) (19)
=loop(self, X, Y) =new case_res (
    // case head
    TrPIexp(self, {X, hello, Y}) ||
    // case body
    TrPIexp(case_res, receive
        {monday, hello, swimming} -> go_for_swim;
        {Weekend, hi, rest} -> take_rest;
        Undefined -> sleep
    end) )

```

```

((24 with res -> case_res) ->(5) (3)), ((14) ->(26D))
=new case_res (
    // case head
    ( case_res'<X, hello, Y>.nil ||
    // case body
    (
        // first match
        case_res(input_pat1, input_pat2, input_pat3).[input_pat1=monday]
        [input_pat2=hello] [input_pat3=swimming] fun_case_res'<go_for_swim>.nil +
        // second match
        case_res(Weekend, input_pat4, input_pat5).[input_pat4=hi] [input_pat5=rest]
        fun_case_res'<take_rest>.nil +
        // third match
        case_res(Undefined). fun_case_res'<sleep>.nil
    ) ) )

```

Finally, **BNSRAV**= { Weekend, Undefined }

5.6.1(c) The π -Model

From Section 5.6.1(b), the π -model of Program 5.7 can be written as follows:

```

main() = new self(start(self))

start(self) = new exp1_res, exp2_res, lpid, loop_res ( exp1_res'<monday>.nil ||
exp1_res(Weekday). ( exp2_res'<swimming>.nil || exp2_res(Work).( loop(lpid,
Weekday, Work) || loop_res(dummy) ) ) )

```

```

loop(self, X, Y)=new case_res (
  // case head
  ( case_res'<X, hello, Y>.nil ||
    // case body
    (
      // first match
      case_res(input_pat1, input_pat2, input_pat3).[input_pat1=monday]
      [input_pat2=hello] [input_pat3=swimming] fun_case_res'<go_for_swim>.nil +
        // second match
        case_res(Weekend, input_pat4, input_pat5).[input_pat4=hi] [input_pat5=rest]
        fun_case_res'<take_rest>.nil +
          // third match
          case_res(Undefined). fun_case_res'<sleep>.nil
        ) ) )
) ) )

```

5.6.1(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))
```

```

(substituting RHS of process start(self))
=new exp1_res, exp2_res, lpid, loop_res ( exp1_res'<monday>.nil ||
exp1_res(Weekday). ( exp2_res'<swimming>.nil || exp2_res(Work). ( loop(lpid,
Weekday, Work) || loop_res(dummy) ) ) )

```

```

(react on exp1_res) -> (react on exp2_res) (in this way name Weekday and Work are
replaced with atoms monday and swimming respectively)
=new self, exp1_res, exp2_res, lpid, loop_res (nil || (nil || ( loop(lpid, monday,
swimming) || loop_res(dummy) ) ) )

```

```

(substituting RHS of process loop(self, X, Y) with self->lpid, X->monday,
Y->swimming)
=new self, exp1_res, exp2_res, lpid, loop_res, case_res(nil || (nil || (
  // case head
  ( case_res'<monday, hello, swimming>.nil ||
    // case body
    (

```

```

    // first match
case_res(input_pat1, input_pat2, input_pat3).[input_pat1=monday]
[input_pat2=hello] [input_pat3=swimming] fun_case_res'<go_for_swim>.nil +
    // second match
case_res(Weekend, input_pat4, input_pat5).[input_pat4=hi] [input_pat5=rest]
fun_case_res'<take_rest>.nil +
    // third match
case_res(Undefined). fun_case_res'<sleep>.nil
)) loop_res(dummy) ))

```

In *case head*, a tuple of message $\{monday, hello, swimming\}$ is sent along channel *case_res*. There are two possibilities in *case body* to have a *receive* action along channel *case_res*. We consider that the tuple of message is sent from *case head* and the tuple *receive action* of first match in *case body* receives that tuple.

```

(react on case_res, with first match )
=new self, exp1_res, exp2_res, lpid, loop_res, case_res(nil || (nil || (
    // case head
    ( nil ||
    // case body
    (
        // first match
        [monday=monday] [hello=hello] [swimming=swimming
        fun_case_res'<go_for_swim>.nil
    )) loop_res(dummy) ))

```

Name matching ($[monday=monday][hello=hello][swimming=swimming]$) is found and as a result atom *go_for_swim* is sent along the result channel *fun_case_res* of loop process. This is the usual situation where accurate expected result is found as of PIERlang Program 5.7. If the *tuple receive action* of the second match is communicated with the *case head* instead of the *tuple receive action* of the first match then name mismatching will be found and there will be a possibility of deadlock in the system. We consider that such deadlocks will be handled in π -calculus by looking for further possibility of successfully matching.

5.7 An Approach to Improve Send Rule (25)

The translation mappings rules for *send* and *receive* expressions with tuples discussed Sections 5.4 and 5.5 must meet the equality semantics of *send-receive* expressions

with respect to the size of tuples used i.e., if a tuple of size n is sent with *rule* (25), then in receiving side (Match rule (26D)), there must be at least one matching tuple pattern of size n . But in PIErlang or in Erlang, there may be different situations. Let us consider the following Program 5.8.

```

start() ->
    Loop_Pid=spawn(loop, []),
    Loop_Pid ! {self(), hello}.
loop() ->
    receive
        {From, Message, request} -> do_works;
        stop -> terminate;
        Y -> sleep
    end.

```

Program 5.8 An Unusual Program

5.7. 1 PIErlang Program 5.8: Send rule(25) is Insufficient

In this section, rule (25) is found insufficient to meet PIErlang semantics.

5.7.1(a) Execution in PIErlang Compiler

The execution mechanism of Program 5.8 in PIErlang compiler is shown in Figure 5.7. As usual spawn function in start function invokes the loop function to be executed in parallel and then start process(start function) sends the tuple of message $\{self(), hello\}$ to the loop process(loop function).

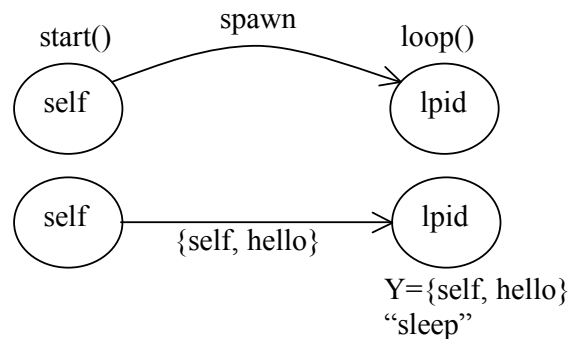


Figure 5.7 Schematic diagram of Program 5.8

In loop process, there is no pattern in any of matches that can match such a tuple of size 2. However, as there is a variable pattern in the last match, and as variable pattern can match with any term, sent tuple message will be matched with the pattern variable Y and consequently atom *sleep* will be returned as the result of the loop process.

5.7.1(b) Translation in the π -calculus

According to rule (21),

$\text{main}() = \text{new self}(\text{start}(\text{self}))$

$\text{TrPI}_{\text{funder}}(\text{self}, \text{start}() \rightarrow$
 $\text{Loop_Pid} = \text{spawn}(\text{loop}, []),$
 $\text{Loop_Pid} ! \{\text{self}(), \text{hello}\})$

Initially, \mathbf{BNSRAV} is empty i.e. $\mathbf{BNSRAV} = \{ \}$

(20A) \rightarrow (11) \rightarrow (25) (10A) (22) (5) (3)
 $= \text{start}(\text{self}) = \text{new lpid}, p, \text{loop_res}, \text{start_send_res}(p' \langle \text{lpid} \rangle . \underline{\text{nil}} \parallel \text{loop}(\text{lpid}) \parallel$
 $\text{loop_res}(\text{dummy}). \underline{\text{nil}} \parallel p(\mathbf{Loop_Pid}). (\text{Loop_Pid}' \langle \text{self}, \text{hello} \rangle . \underline{\text{nil}} \parallel$
 $\text{start_send_res}' \langle \text{self}, \text{hello} \rangle . \underline{\text{nil}})$

Finally, $\mathbf{BNSRAV} = \{ \mathbf{Loop_Pid} \}$

$\text{TrPI}_{\text{funder}}(\text{self}, \text{loop}() \rightarrow$
 receive
 $\{ \text{From}, \text{Message}, \text{request} \} \rightarrow \text{do_works};$
 stop \rightarrow terminate;
 $Y \rightarrow \text{sleep}$
 end.)

Initially, \mathbf{BNSRAV} is empty i.e. $\mathbf{BNSRAV} = \{ \}$

(20A) \rightarrow (14) \rightarrow (26D) \rightarrow (5)(3)(4)
 $= \text{loop}(\text{self}) = \text{self}(\mathbf{From}, \mathbf{Message}, \text{input_pat3}). [\text{input_pat3} = \text{request}]$
 $\text{loop_res}' \langle \text{do_works} \rangle . \underline{\text{nil}} + \text{self}(\text{input_pat1}). [\text{input_pat1} = \text{stop}]$
 $\text{loop_res}' \langle \text{terminate} \rangle . \underline{\text{nil}} + \text{self}(\mathbf{Y}). \text{loop_res}' \langle \text{sleep} \rangle . \underline{\text{nil}}$

Finally, $\mathbf{BNSRAV} = \{ \text{From}, \text{Message}, \mathbf{Y} \}$

5.7.1(c) The π -Model

From Section 5.7.1(b), the π -model of Program 5.8 can be written as follows:

```
main()=new self(start(self))

start(self) = new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
  loop_res(dummy).nil || p(Loop_Pid).( Loop_Pid'<self, hello>.nil ||
  start_send_res'<self, hello>.nil ) )

loop(self)=self(From, Message, input_pat3).[input_pat3=request]
  loop_res'<do_works>.nil + self(input_pat1).[input_pat1=stop]
  loop_res'<terminate>.nil + self(Y).loop_res'<sleep>.nil
```

5.7.1(d) Observing Behavior in π -calculus: Send rule (25) cannot provide full PIErlang Receive Semantics

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))

(substituting RHS of process start(self))
=>new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
  loop_res(dummy).nil || p(Loop_Pid).( Loop_Pid'<self, hello>.nil ||
  start_send_res'<self, hello>.nil ) )
```

```
(react on p)->(Omitting nil process)
=>new lpid, p, loop_res, start_send_res(
  // loop process
  loop(lpid) || loop_res(dummy).nil ||
  // start process
  ( lpid'<self, hello>.nil || start_send_res'<self, hello>.nil ) )
```

```
(substituting RHS of process loop(self) with self -> lpid)
=>new lpid, p, loop_res, start_send_res(
  // loop process
  (lpid(From, Message, input_pat3).[input_pat3=request]
  loop_res'<do_works>.nil + lpid(input_pat1).[input_pat1=stop]
  loop_res'<terminate>.nil + lpid(Y).loop_res'<sleep>.nil) || loop_res(dummy).nil
  || // start process
  ( lpid'<self, hello>.nil || start_send_res'<self, hello>.nil ) )
```


Now we see that start process sends a tuple of message(tuple size 2) to the loop process with $lpid' < self, hello > .\underline{nil}$. But in loop process, there is no subprocess that can have a react (*receive action* along channel $lpid$) on channel $lpid$ for tuple size 2, consequently, the system will be in an infinity deadlock state. According to the PIERlang semantics, this sent tuple of message should be received with the 3rd subprocess of the loop process (3rd match $lpid(Y).loop_res' < sleep > .\underline{nil}$). As a result, atom *sleep* should be returned as the result of the loop process. We have found that this problem has been arisen due to the insufficiency of *send* rule (25). Therefore, a modification of this rule is required(cf. Section 5.7.2).

5.7. 2 A Modification to Send Rule (25)

To overcome the problem of the semantic matching between PIERlang and π -calculus that arises in Section 5.7.1(d) for Program 5.8, rule (25) is modified as follows:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, A \ ! \{A_1, A_2, \dots, A_n\}) = & ((\text{TrPI}_{\text{arg}}(A))' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{nil} + \\ & (\text{TrPI}_{\text{arg}}(A))' < \text{unknownTuple} > .\underline{nil}) \parallel (\text{res}' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{nil} + \\ & \text{res}' < \text{unknownTuple} > .\underline{nil}) \end{aligned} \quad \text{-(25A)}$$

Along with the previous definition, two more subprocesses are added using non-deterministic choices. Whenever a tuple-based message is sent, a name *unknownTuple* is also sent non-deterministically so that the problem arisen in Section 5.7.1(d) can be solved. The reason of sending the tuple of message along the *res* channel has already been mentioned Section 5.4. For the same reason as of first non-determinism, name *unknownTuple* is also sent along the *res* channel as the second non-determinism.

5.7. 3. PIERlang Program 5.8: Send Rule(25A) Sounds Correct

In this section, rule (25A) is used for translation mapping Program 5.8 and found that translated model with rule (25A) overcomes the problem that arisen in Section 5.7.1(d).

5.7.3(a) Execution in PIERlang Compiler

The execution mechanism of Program 5.8 in PIERlang compiler is described in Section 5.7.1(a).

5.7.3(b) Translation in the π -calculus

According to rule (21),

```
main()=new self(start(self))
```

```
TrPIfunder(self, start() ->
    Loop_Pid=spawn(loop, []),
    Loop_Pid ! {self(), hello})
```

Initially, **BNSRAV** is empty i.e. **BNSRAV**=**{ }**

```
(20A)->(11)->(25A) (10A) (22) (5) (3)
=start(self) = new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). ( (Loop_Pid'<self, hello>.nil +
Loop_Pid'<unknownTuple>.nil) || (start_send_res'<self, hello>.nil +
start_send_res'<unknownTuple>.nil ) )
```

Finally, **BNSRAV**=**{Loop_Pid }**

Translation for the loop process is same as of Section 5.7.1(b).

5.7.3(c) The π -Model

From Section 5.7.3(b), the modified π -model of Program 5.8 can be written as follows:

```
main()=new self(start(self))
```

```
start(self) = new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). ( (Loop_Pid'<self, hello>.nil +
Loop_Pid'<unknownTuple>.nil) || (start_send_res'<self, hello>.nil +
start_send_res'<unknownTuple>.nil ) )
```

```
loop(self)=self(From, Message, input_pat3).[input_pat3=request]
    loop_res'<do_works>.nil + self(input_pat1).[input_pat1=stop]
    loop_res'<terminate>.nil + self(Y).loop_res'<sleep>.nil
```

5.7.3(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))
```

(substituting RHS of process *start(self)*)

```
=>new lpid, p, loop_res, start_send_res( p'<lpid>.nil || loop(lpid) ||
    loop_res(dummy).nil || p(Loop_Pid). ( (Loop_Pid'<self, hello>.nil +
    Loop_Pid'<unknownTuple>.nil) || (start_send_res'<self, hello>.nil +
    start_send_res'<unknownTuple>.nil ) )
```

(react on **p**) -> (Omitting nil process)

```
=>new lpid, p, loop_res, start_send_res(
    // loop process
    loop(lpid) || loop_res(dummy).nil
    || // start process
    ((lpid'<self, hello>.nil + lpid'<unknownTuple>.nil) || (start_send_res'<self,
    hello>.nil + start_send_res'<unknownTuple>.nil ) )
```

(substituting RHS of process *loop(self)* with **self** -> **lpid**)

```
=>new lpid, p, loop_res, start_send_res(
    // loop process
    (lpid(From, Message, input_pat3).[input_pat3=request]
    loop_res'<do_works>.nil + lpid(input_pat1).[input_pat1=stop]
    loop_res'<terminate>.nil + lpid(Y).loop_res'<sleep>.nil) || loop_res(dummy).nil
    || // start process
    ((lpid'<self, hello>.nil + lpid'<unknownTuple>.nil) || (start_send_res'<self,
    hello>.nil + start_send_res'<unknownTuple>.nil ) )
```

In contrast to Section 5.7.1(d), here we see that name *unknownTuple* is also sent along channel *lpid* and in loop process, subprocess *lpid(Y).loop_res'<sleep>.nil* can now have a react on *lpid*.

(react on **lpid**)

```
=>new lpid, p, loop_res, start_send_res(
    // loop process
    (loop_res'<sleep>.nil) || loop_res(dummy).nil
    || // start process
    ((nil) || (start_send_res'<self,
    hello>.nil + start_send_res'<unknownTuple>.nil ) )
```

As a result of applying react on channel *lpid* atom *sleep* is sent along the result channel of the loop process thus, meeting the PIERlang semantics as of Section 5.7.1(a) and solving the problem of Section 5.7.1(d).

5.8 An Approach to Improve Tuple Expression Rule (24)

The problem cited in Section 5.7 for the case of send rule(25), could also be arisen in the case of tuple rule (24) of Section 5.3, especially, when a tuple expression is used as the *case head* of the *case* expression. If the sent tuple message of Program 5.8 is used in *case head* then the same problem will be found as of Section 5.7.1(d). To overcome this problem, rule (24) is modified as follows:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \{A_1, A_2, \dots, A_n\}) = & \text{res}' \langle \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) \rangle .\underline{\text{nil}} + \\ & \text{res}' \langle \text{unknownTuple} \rangle .\underline{\text{nil}} \end{aligned} \quad \text{-(24A)}$$

5.9 An Alternative Approach for Match Rule (26D)

In Program 5.8, a tuple of size 2 is sent to the loop process. On the other hand, loop process is not ready to receive a tuple of size 2, rather it can receive a tuple of size 3 or a single atom *stop* or any message (as variable is used as pattern). However, we see that there is some sort of similarity between the sending message and the pattern of the match(s) with respect to their types. A tuple is sent by the start process(sender process) and in loop process(receiver process) there is also a tuple pattern in the first match. We consider that sent message can have a pattern matching with the pattern of the first match with respect to their types. Considering such situations, we have proposed an alternative approach for rule (26D) as follows:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = & \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_patti}, \dots) \dots [\text{input_patti} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) \\ \quad + \text{self}(\text{Tuple}).[\text{Tuple} = \text{unknownTuple}] \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \quad \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNSRAV} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \quad \text{variable } X \text{ and name } X \notin \text{BNSRAV} \end{array} \right. \end{aligned} \quad \text{-(26E)}$$

In rule(26E), we see that whenever a tuple is used as pattern in the matches, a *monadic receive action* is used with non-deterministic choice to receive a name in *Tuple* and if the received name is *unknownTuple*, the *body* expression associated with the tuple pattern will be evaluated. We already know that name *unknownTuple* can be sent by the send rule (25A) or tuple expression rule (24A). However, using this rule there is possibility of breaking the semantics of PIERlang in translated π -model. Therefore, we will not use rule (26E) in the subsequent translation mappings.

5.10 TrPIs at a Glance

Only the most promising final outcomes of the rules discussed in this chapter are mentioned in this section.

TrPI_{exp}: Name X Expression -> Process

$$\text{TrPI}_{\text{exp}}(\text{self}, \text{self}()) := \text{res}' < \text{self} > .\underline{\text{nil}} \quad -(23)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \{A_1, A_2, \dots, A_n\}) = & \text{res}' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} + \\ & \text{res}' < \text{unknownTuple} > .\underline{\text{nil}} \end{aligned} \quad -(24A)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, A ! \{A_1, A_2, \dots, A_n\}) = & ((\text{TrPI}_{\text{arg}}(A))' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} + \\ & (\text{TrPI}_{\text{arg}}(A))' < \text{unknownTuple} > .\underline{\text{nil}}) \parallel (\text{res}' < \text{TrPI}_{\text{arg}}(A_1), \dots, \text{TrPI}_{\text{arg}}(A_n) > .\underline{\text{nil}} + \\ & \text{res}' < \text{unknownTuple} > .\underline{\text{nil}}) \end{aligned} \quad -(25A)$$

TrPI_{match}: Name X Match -> Process

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNSRAV} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \text{variable } X \text{ and name } X \notin \text{BNSRAV} \end{array} \right. \quad -(26D) \end{aligned}$$

Where $\text{BNSRAV} = \{ X \mid y(\dots, X, \dots) \text{ is a receive action in } \pi\text{-calculus and } X \text{ is a variable in PIERlang function for which } X \text{ is considered as a name in } \pi\text{-calculus.} \}$

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) \\ \quad + \text{self}(\text{Tuple}).[\text{Tuple} = \text{unknownTuple}] \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNSRAV} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \text{variable } X \text{ and name } X \notin \text{BNSRAV} \end{array} \right. \quad -(26E) \end{aligned}$$

TrPI_{arg}: Argument -> Name

$$\text{TrPI}_{\text{arg}}(\text{self}()) := \text{self} \quad -(22)$$

Chapter 6

Mapping Nested Tuples, Lists and Arithmetic Expressions

In this chapter, the uses of nested tuples, lists and arithmetic expressions are presented and for each of the syntactic constructs a translation mapping rule in π -calculus is provided. Additionally, one program(using these constructs) has been used to get corresponding system model in π -calculus by applying translation mapping rules. Furthermore, it is shown that gained π -calculus model shows the same behavior as it could be expected from its corresponding Erlang program.

Table of Contents \Rightarrow	6.1 PIERlang-02 Syntax	-132
	6.2 Data Types	-133
	6.3 Arithmetic Expressions	-134
	6.4 Lists	-135
	6.5 Nested Tuples	-135
	6.6 Send Expression	-135
	6.7 Matches	-136
	6.8 PIERlang Program 6.1: A Different Approach	-136
	6.8(a) Execution in PIERlang Compiler	-137
	6.8(b) Translation in the π -calculus	-137
	6.8(c) The π -Model	-139
	6.8(d) Observing Behavior in π -calculus	-139

6.1 PIERlang-02 Syntax

In Chapter 5, we have discussed the translation mapping based on the PIERlang-01 which is an extension of PIERlang-00 for supporting uses of non-nested tuples as an expression, message of *send* expression and *patterns* of matches of *receive* and *case* expressions. In this chapter, we have modified PIERlang-01 for supporting nested tuples and we have renamed PIERlang-01 with PIERlang-02.

Program	$P ::= F+ ; E$	
Function Definition	$F ::= f(X_1, X_2, \dots, X_n) \rightarrow E$;n>=0
Expression	$E ::= n \mid a \mid X$	
	$\mid \{U_1, \dots, U_n\}$;n>=0
	$\mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2$	
	$\mid [A_1 \mid A_2] [A_1, \dots, A_n]$; n>=0
	$\mid X = E_1, E_2 \mid X = E \mid E_1, E_2$	
	$\mid \text{self}() \mid f(U_1, \dots, U_n) \mid \text{spawn}(f, [U_1, \dots, U_n])$;n>=0
	$\mid A ! V \mid A ! \{U_1, \dots, U_n\}$;n>=0
	$\mid \text{receive } M_1; \dots ; M_n \text{ end} \mid \text{case } E \text{ of } M_1; \dots ; M_n \text{ end}$;n>=0
Match	$M ::= V \rightarrow E \mid \{P_1, \dots, P_n\} \rightarrow E$;n>=0
Pattern	$P ::= n \mid a \mid X \mid \{A_1, \dots, A_n\} \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$;n>=0
Argument	$A ::= n \mid a \mid X \mid \text{self}()$	
Argument	$U ::= n \mid a \mid X \mid \text{self}() \mid \{A_1, \dots, A_n\} \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$;n>=0
Argument	$V ::= n \mid a \mid X \mid \text{self}() \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$;n>=0
	$n \in \text{Numbers} ;$	
	$a, f \in \text{Atoms} ;$	
	$X, X_1, \dots, X_n \in \text{Variables}$	

Figure 6.1 PIERlang-02 (added syntactic constructs are marked with **boldface**)

In this modified version, Lists are also used in various contexts. Syntactic constructs of PIERlang-02 are presented in Figure 6.1.

In the following sections, we will gradually discuss the corresponding π -calculus mapping for each of the syntactic constructs of PIERlang-02. Unless otherwise modified or stated clearly previous translation mapping rules (rules (1) to (26E)) of Chapter 4 and Chapter 5 will be used here when necessary.

6.2 Data Types

In Section 4.2, translation mapping rules for *Data Types* have been discussed. Although the translation mapping rules (3) and (4) for atoms are used unchanged, rules (1) & (2) for integers and (1A) and (2A) floats have been modified as follows:

$$\text{TrPI}_{\text{arg}}(n)=\text{unknownInteger} \quad -(27)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, n)=\text{res}'<\text{TrPI}_{\text{arg}}(n)>.\underline{\text{nil}}$$

$$\begin{aligned} (27) \\ = \text{res}'<\text{unknownInteger}>.\underline{\text{nil}} \end{aligned} \quad -(28)$$

$$\text{TrPI}_{\text{arg}}(f1)=\text{unknownFloat} \quad -(27A)$$

$$\text{TrPI}_{\text{exp}}(\text{self}, f1)=\text{res}'<\text{TrPI}_{\text{arg}}(f1)>.\underline{\text{nil}}$$

$$\begin{aligned} (27A) \\ = \text{res}'<\text{unknownFloat}>.\underline{\text{nil}} \end{aligned} \quad -(28A)$$

In Chapter 4, integers and floats have been considered in the same way where their translations to π -calculus is just a name *unknown* while used as arguments and name *unknown* is passed along the *res* channel while used as expressions. Here integers and floating points numbers are treated separately as we see in the above rules.

6.3 Arithmetic Expressions

In this section, we have introduced the uses of mathematical operators (+, -, * and /) on atomic numerical expressions as follows:

$$\begin{aligned} \text{TrPI}_{\text{arg}}(E_1 + E_2)= \text{TrPI}_{\text{arg}}(E_1 - E_2)= \text{TrPI}_{\text{arg}}(E_1 * E_2)=\text{TrPI}_{\text{arg}}(E_1 / E_2)=\text{unknownInteger}, \\ \text{if } \text{TrPI}_{\text{arg}}(E_1)=\text{unknownInteger} \text{ and } \text{TrPI}_{\text{arg}}(E_2)=\text{unknownInteger} \end{aligned} \quad -(29)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(E_1 + E_2)= \text{TrPI}_{\text{exp}}(E_1 - E_2)= \text{TrPI}_{\text{exp}}(E_1 * E_2) \\ = \text{TrPI}_{\text{exp}}(E_1 / E_2)=\text{res}'<\text{unknownInteger}>.\underline{\text{nil}} \\ \text{if } \text{TrPI}_{\text{arg}}(E_1)=\text{unknownInteger} \text{ and } \text{TrPI}_{\text{arg}}(E_2)=\text{unknownInteger} \end{aligned} \quad -(30)$$

$$\begin{aligned} \text{TrPI}_{\text{arg}}(E_1 + E_2)= \text{TrPI}_{\text{arg}}(E_1 - E_2)= \text{TrPI}_{\text{arg}}(E_1 * E_2)=\text{TrPI}_{\text{arg}}(E_1 / E_2)=\text{unknownFloat}, \\ \text{if } \text{TrPI}_{\text{arg}}(E_1)=\text{unknownFloat} \text{ and/or } \text{TrPI}_{\text{arg}}(E_2)=\text{unknownFloat} \end{aligned} \quad -(29A)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(E_1 + E_2)= \text{TrPI}_{\text{exp}}(E_1 - E_2)= \text{TrPI}_{\text{exp}}(E_1 * E_2) \\ = \text{TrPI}_{\text{exp}}(E_1 / E_2)=\text{res}'<\text{unknownFloat}>.\underline{\text{nil}}, \\ \text{if } \text{TrPI}_{\text{arg}}(E_1)=\text{unknownFloat} \text{ and/or } \text{TrPI}_{\text{arg}}(E_2)=\text{unknownFloat} \end{aligned} \quad -(30A)$$

6.4 Lists

From PIERlang-02 syntax (Figure 6.1) above, it is found that Lists are used as simple expressions, as arguments of *function calls* and *spawn calls*, as an atomic message of *send* expression, as an element of the tuple message of *send* expression, as an atomic *pattern* of the matches of *receive* and *case* expressions and as an element of the tuple *pattern* of *case* and *receive* expressions. Lists are treated as arguments except the case where they are used as simple expressions. We have not taken into consideration to support the full semantics of Lists in translated π -calculus system. Therefore, we have considered that if there is any instance of Lists, it will be translated to a name in π -calculus while used as arguments and arguments translation will be sent along the *res* channel while used as simple expression(s) as follows:

$$\text{TrPI}_{\text{arg}}([]) = \text{emptyList} \quad \text{-(31)}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, []) = \text{res}' \langle \text{emptyList} \rangle . \underline{\text{nil}} \quad \text{-(32)}$$

$$\text{TrPI}_{\text{arg}}([A_1, \dots, A_n]) = \text{unknownList} \quad \text{-(31A)}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, [A_1, \dots, A_n]) = \text{res}' \langle \text{unknownList} \rangle . \underline{\text{nil}} \quad \text{-(32A)}$$

$$\text{TrPI}_{\text{arg}}([A_1 \mid A_2]) = \text{unknownList} \quad \text{-(31B)}$$

$$\text{TrPI}_{\text{exp}}(\text{self}, [A_1 \mid A_2]) = \text{res}' \langle \text{unknownList} \rangle . \underline{\text{nil}} \quad \text{-(32B)}$$

6.5 Nested Tuples

In Chapter 5, we have discussed the uses of non-nested tuples as simple expression, as the message of *send* expression and *pattern* of matches of *receive* and *case* expressions where the tuple element(s) could be only number(s), atom(s), variable(s) and/or PID self(). In this section, our intention is to use nested tuple (at least 1-nested tuple) in the contexts where non-nested tuples are used in Chapter 5.

$$\text{TrPI}_{\text{arg}}(\{A_1, \dots, A_n\}) = \text{unknownTuple} \quad \text{-(33)}$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \{U_1, \dots, U_n\}) = & \text{res}' \langle \text{TrPI}_{\text{arg}}(U_1), \dots, \text{TrPI}_{\text{arg}}(U_n) \rangle . \underline{\text{nil}} + \\ & \text{res}' \langle \text{unknownTuple} \rangle . \underline{\text{nil}} \end{aligned} \quad \text{-(24B)}$$

6.6 Send Expression

In Section 4.4, *send* expression with atomic arguments has been discussed. In Sections 5.4 and 5.7, tuple based *send* expression has been discussed in details where atomic

elements such as numbers, atoms, variables and `self()` are used as the elements of the tuple message. In this section, we have added lists and tuples as atomic message or as elements of the tuple message used in the *send* expression. The modified rules are as follows:

$$\text{TrPI}_{\text{exp}}(\text{self}, A ! V) = (\text{TrPI}_{\text{arg}}(A))' < \text{TrPI}_{\text{arg}}(V) > .\underline{\text{nil}} \parallel \text{res}' < \text{TrPI}_{\text{arg}}(V) > .\underline{\text{nil}} \quad -(9A)$$

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, A ! \{U_1, \dots, U_n\}) = & ((\text{TrPI}_{\text{arg}}(A))' < \text{TrPI}_{\text{arg}}(U_1), \dots, \text{TrPI}_{\text{arg}}(U_n) > .\underline{\text{nil}} + \\ & (\text{TrPI}_{\text{arg}}(A))' < \text{unknownTuple} > .\underline{\text{nil}}) \parallel \\ & (\text{res}' < \text{TrPI}_{\text{arg}}(U_1), \dots, \text{TrPI}_{\text{arg}}(U_n) > .\underline{\text{nil}} + \\ & \text{res}' < \text{unknownTuple} > .\underline{\text{nil}}) \end{aligned} \quad -(25B)$$

6.7 Matches

Tuple-based match rules (26D) and (26E) are also valid here with additional support of allowing tuples and lists as the elements of the tuple pattern. Thus rule (26D) is renamed as (26F) with tuple element $P ::= n \mid a \mid X \mid \{A_1, \dots, A_n\} \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\} \rightarrow E) = \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati} = \text{TrPI}_{\text{arg}}(P_i)] \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \\ \text{if } P_i \in \{\text{Atoms}, \text{Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name } P_i \in \text{BNSRAV} \\ \\ \text{self}(\dots, X, \dots) \dots \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \text{ where } P_i \text{ is a} \\ \text{variable } X \text{ and name } X \notin \text{BNSRAV} \end{array} \right. \quad -(26F) \end{aligned}$$

Where $\text{BNSRAV} = \{ X \mid y(\dots, X, \dots) \text{ is a receive action in } \pi\text{-calculus and } X \text{ is a variable in PIERlang function for which } X \text{ is considered as a name in } \pi\text{-calculus.} \}$

Single pattern match rule $\text{TrPI}_{\text{match}}(\text{self}, V \rightarrow E)$ is developed to fit with the added syntactic constructs of *send* rule (25B) as follows:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, V \rightarrow E) = & \text{self}(\text{input_pat}).[\text{input_pat} = \text{TrPI}_{\text{arg}}(V)] \text{TrPI}_{\text{exp}}(\text{self}, E) \\ \text{Where } V ::= & n \mid a \mid \text{self}() \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n] \quad ; n \geq 0 \quad -(34) \end{aligned}$$

6.8 PIERlang Program 6.1: A Different Approach

Let us consider a simple example as follows:

```
start() ->
    Loop_Pid = spawn(loop, []),
    Loop_Pid ! {[X | Allowed], hello, {do, die}}.
loop() ->
```

```

receive
  { [Z | Worked], hello, {Die, Do}} -> go_market;
  stop -> terminate;
  Y -> sleep
end.

```

Program 6.1. Simple Message passing example

6.8(a) Execution in PIERlang Compiler

In PIERlang-02, we assume that compiler can perform pattern matching against the types of the elements of nested tuple-based terms. As usual, *spawn* call causes the start process and the loop process to be executed in parallel. A nested tuple message {[X, Allowed], hello, {do, die}} is sent to the loop process. In the loop process, there is no such matching pattern for having a successful match with the sent tuple message. However, PIERlang then tries to have similarity between the types of the elements of the tuple for pattern matching. Consequently, PIERlang will consider that a tuple message(*{list, hello, tuple}*) is sent to the loop process where the first element of the tuple is a *list*, second is an atom *hello* and third element is again a *tuple*. PIERlang compiler will try to have such a tuple in loop process matches and if there is any such tuple pattern, a successful pattern matching will be occurred. We see that in the loop process, there is a pattern {[Z | Worked], hello, {Die, Do}} which will be treated by PIERlang as *{list, hello, tuple}*. As a result, a successful pattern matching will be found and *go_market* will be returned as a result of the loop process.

6.8(b) Translation in the π -calculus

According to rule (21),

main()=new self(start(self))

TrPI_{exp}(self, start() ->

Loop_Pid=spawn(loop, []),

Loop_Pid ! {[X | Allowed], hello, {do, die}})

Initially, **BNSRAV** is empty i.e. **BNSRAV**= { }

(20A)->(11)->(10A)

=start(self)=new lpid, p, loop_res (p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
p(Loop_Pid).(**Loop_Pid ! {[X | Allowed], hello, {do, die}}**))

(25B)

```
=new lpid, p, loop_res ( p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
  p(Loop_Pid).( ( Loop_Pid'<TrPI_arg([X | Allowed]), TrPI_arg(hello), TrPI_arg( {do,
  die})>.nil + Loop_Pid'<unknownTuple>.nil ) || ( res'<TrPI_arg([X | Allowed]),
  TrPI_arg(hello), TrPI_arg( {do, die})>.nil + res'<unknownTuple>.nil ) ) )
```

NSRAV={ Loop_Pid }

(31A) (3) (33)

```
=new lpid, p, loop_res ( p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
  p(Loop_Pid).( ( Loop_Pid'<unknownList, hello, unknownTuple>.nil +
  Loop_Pid'<unknownTuple>.nil ) || ( res'<unknownList, hello, unknownTuple>.nil +
  res'<unknownTuple>.nil ) ) )
```

BNSRAV={Loop_Pid }

TrPI_{exp}(self, loop() ->

receive

{[Z | Worked], hello, {Die, Do}} -> go_market;

stop -> terminate;

Y -> sleep

end)

Initially, **BNSRAV** is empty i.e. **BNSRAV**={ }

(20A)->(14)

```
=loop(self)=TrPImatch( self, {[Z | Worked], hello, {Die, Do}} -> go_market) +
  TrPImatch(self, stop -> terminate) + TrPImatch(self, Y -> sleep)
```

(26F)

```
= self(input_pat1, input_pat2, input_pat3).[input_pat1=TrPIarg([Z | Worked])]
  [input_pat2=TrPIarg(hello)] [input_pat2=TrPIarg( {Die, Do})]
  loop_res'<go_market>.nil + self(input_pat4).[input_pat4=stop]
  loop_res'<terminate>.nil + self(Y).loop_res'<sleep>.nil
```

BNSRAV={Y}

(31A) (3) (33)

```
= self(input_pat1, input_pat2, input_pat3).[input_pat1=unknownList] [input_pat2=
hello] [input_pat3=unknownTuple] loop_res'<go_market>.nil +
self(input_pat4).[input_pat4= stop] loop_res'<terminate>.nil + self(Y).
loop_res'<sleep>.nil
```

BNSRAV={Y}

6.8(c) The π -Model

From Section 6.8(b), the π -model of Program 6.1 can be written as follows:

```
main()=new self(start(self))

start(self)= new lpid, p, loop_res ( p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
  p(Loop_Pid).( ( Loop_Pid'<unknownList, hello, unknownTuple>.nil +
    Loop_Pid'<unknownTuple>.nil ) || ( res'<unknownList, hello, unknownTuple>.nil +
    res'<unknownTuple>.nil ) ) )

loop(self)= self(input_pat1, input_pat2, input_pat3).[input_pat1=unknownList]
  [input_pat2= hello] [input_pat3=unknownTuple] loop_res'<go_market>.nil +
  self(input_pat4).[input_pat4= stop] loop_res'<terminate>.nil + self(Y).
  loop_res'<sleep>.nil
```

6.8(d) Observing Behavior in π -calculus

To observe the model behavior in π -calculus, we have to start from the *main()* process.

```
main()=new self(start(self))

(process call start(self))
=new self, lpid, p, loop_res( p'<lpid>.nil || loop(lpid) || loop_res(dummy).nil ||
  p(Loop_Pid).( (Loop_Pid'<unknownList, hello, unknownTuple>.nil +
    Loop_Pid'<unknownTuple>.nil ) || ( res'<unknownList, hello,
    unknownTuple>.nil + res'<unknownTuple>.nil ) ) )

(react on p) ->(omitting nil process)
=new self, lpid, p, loop_res (
  // loop process
  loop(lpid) || loop_res(dummy).nil ||
  // start process
  ( ( lpid'<unknownList, hello, unknownTuple>.nil + lpid'<unknownTuple>.nil ) || (
    res'<unknownList, hello, unknownTuple>.nil + res'<unknownTuple>.nil ) ) )
```

```

(process call loop(lpid) with self -> lpid)
(*)=new self, lpid, p, loop_res (
    // loop process
    ( lpid(input_pat1, input_pat2, input_pat3).[input_pat1=unknownList]
      [input_pat2=hello] [input_pat3=unknownTuple] loop_res'<go_market>.nil +
      lpid(input_pat4).[input_pat4= stop] loop_res'<terminate>.nil +
      lpid(Y).loop_res'<sleep>.nil ) || loop_res(dummy).nil ||
    // start process
    ( ( lpid'<unknownList, hello, unknownTuple>.nil + lpid'<unknownTuple>.nil )
      || ( res'<unknownList, hello, unknownTuple>.nil + res'<unknownTuple>.nil ) ) )

```

```

(react on lpid)
=new self, lpid, p, loop_res (
    // loop process
    ( [unknownList=unknownList] [hello=hello] [unknownTuple=unknownTuple]
      loop_res'<go_market>.nil +
    // start process
    ( ( nil ) || (
      res'<unknownList, hello, unknownTuple>.nil + res'<unknownTuple>.nil ) ) )

```

As a consequence of successful name matching (*[unknownList = unknownList] [hello = hello] [unknownTuple = unknownTuple]*) *go_market* is sent along the *loop_res* channel (result channel of loop process). In Section 6.8(a), we see the same thing in PIERlang compiler. However, as the choices among matches are non-deterministic, there is also another possibility of having a react on *lpid*.

Again consider from (*) above,

```

=new self, lpid, p, loop_res (
    // loop process
    ( lpid(input_pat1, input_pat2, input_pat3).[input_pat1=unknownList] [input_pat2=
      hello] [input_pat3=unknownTuple] loop_res'<go_market>.nil +
      lpid(input_pat4).[input_pat4= stop] loop_res'<terminate>.nil +
      lpid(Y).loop_res'<sleep>.nil ) || loop_res(dummy).nil ||
    // start process
    ( ( lpid'<unknownList, hello, unknownTuple>.nil + lpid'<unknownTuple>.nil ) ||
      ( res'<unknownList, hello, unknownTuple>.nil + res'<unknownTuple>.nil ) ) )

```

```

(react on lpid, 2nd non-determinism loop process and 3rd of start process)
=new self, lpid, p, loop_res (
    // loop process
    ( loop_res'<sleep>.nil ) || loop_res(dummy).nil ||
    // start process
    ( ( nil ) || (
        res'<unknownList, hello, unknownTuple>.nil + res'<unknownTuple>.nil ) ) )

```

As a result *sleep* is sent along *loop_res* channel thus, breaking PIERlang semantics. An approach to avoid such non-determinism among matches has been presented in Section 4.9 and can also be applied here.

Chapter 7

Mapping Guards

In this chapter, Guards are introduced and translation mapping supporting guards are presented. As guards can be trivially mapped with the name matching/mismatching feature of π -calculus we discussed in the earlier chapters, we have not presented any example using guards in this chapter.

	7.1 PIERlang-03 Syntax	-142
	7.2 Guards	-142
Table of Contents \Rightarrow	7.2.1 Guards in Function Definition	-143
	7.2.2 Guards in Matches	-144
	7.3 IF Expression	-144

7.1 PIERlang-03 Syntax

In Chapter 6, we have discussed the translation mapping based on PIERlang-02 syntactic constructs. In this chapter, PIERlang-02 is enriched to support uses of *Guards* in function definitions and in matches of *receive* and *case* expressions. Moreover, *if* expression is added in which *Guard(s)* is/are used.

7.2 Guards

Guards are conditions which have to be fulfilled before a clause is chosen. The reserved word *when* introduces a guard. Fully guarded clauses can be re-ordered. All variables used in a guards must be bound.

$$\text{TrPI}_{\text{guard}}(\text{self, when } C_1, \dots, C_n) := [C_1] \dots [C_n] \quad ; n \geq 1$$

$$\text{TrPI}_{\text{condition}}(\text{self, } V_1 \text{ Op } V_n) := \text{TrPI}_{\text{arg}}(V_1) \text{ Op } \text{TrPI}_{\text{arg}}(V_n)$$

Program	$P ::= F+ ; E$	
Function Definition	$F ::= f(X_1, X_2, \dots, X_n) \text{ [when } \mathbf{G}] \rightarrow E$	$; n \geq 0$
Expression	$E ::= n \mid a \mid X$	
	$\mid \{U_1, \dots, U_n\}$	$; n \geq 0$
	$\mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2$	
	$\mid [A_1 \mid A_2] [A_1, \dots, A_n]$	$; n \geq 0$
	$\mid X = E_1, E_2 \mid X = E \mid E_1, E_2$	
	$\mid \text{self}() \mid f(U_1, \dots, U_n) \mid \text{spawn}(f, [U_1, \dots, U_n])$	$; n \geq 0$
	$\mid A ! V \mid A ! \{U_1, \dots, U_n\}$	$; n \geq 0$
	$\mid \text{receive } M_1; \dots; M_n \text{ end} \mid \text{case } E \text{ of } M_1; \dots; M_n \text{ end}$	$; n \geq 0$
	$\mid \text{if } \mathbf{G}_1 \rightarrow \mathbf{E}_1; \dots; \mathbf{G}_n \rightarrow \mathbf{E}_n \text{ end}$	$; n \geq 0$
Match	$M ::= V[\text{when } \mathbf{G}] \rightarrow E \mid \{P_1, \dots, P_n\}[\text{when } \mathbf{G}] \rightarrow E$	$; n \geq 0$
Pattern	$P ::= n \mid a \mid X \mid \{A_1, \dots, A_n\} \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$	$; n \geq 0$
Argument	$A ::= n \mid a \mid X \mid \text{self}()$	
Argument	$U ::= n \mid a \mid X \mid \text{self}() \mid \{A_1, \dots, A_n\} \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$	$; n \geq 0$
Argument	$V ::= n \mid a \mid X \mid \text{self}() \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n]$	$; n \geq 0$
Guards	$\mathbf{G} ::= C_1, \dots, C_n$	$; n \geq 0$
Condition	$C ::= V_1 \text{ Op } V_n \text{ where } \text{Op} \in \{=, !=\}$	
	$n \in \text{Numbers} ;$	
	$a, f \in \text{Atoms} ;$	
	$X, X_1, \dots, X_n \in \text{Variables}$	

Figure 7.1 PIERlang-03 (added syntactic constructs are marked with **boldface**)

7.2.1 Guards in Function Definition

When a function definition is augmented with guards, they are placed on the right hand side of the function definition in π -calculus translation. Thus, rule(20) is modified as follows:

$$\text{TrPI}_{\text{fundef}}(\text{self}, f(X_1, \dots, X_n) \text{ [when } \mathbf{G}] \rightarrow E) := \\ f(\text{self}, X_1, \dots, X_n) = \text{TrPI}_{\text{guard}}(\text{self}, \text{when } \mathbf{G}) \text{TrPI}_{\text{exp}}(\text{self}, E) \quad \text{-(20B)}$$

7.2.2 Guards in Matches

In Erlang, a pattern can optionally be augmented with a guard for expressing additional conditions on the term that is to be matched against the pattern. A guard consists of a nonempty sequence of guard tests.

The guard tests have sub expressions which are guard expressions. When compared with expressions both guard tests and guard expressions are syntactically restricted. In PIERlang, full semantics of Guards have not been supported(cf. Figure 7.1). Rule (34) can be modified as follows by supporting guards as follows:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, V[\mathbf{when\ G}]\rightarrow E) &= \text{self}(\text{input_pat}).[\text{input_pat}=\text{TrPI}_{\text{arg}}(V)] \\ &\quad \mathbf{TrPI}_{\text{guard}}(\text{self}, \mathbf{when\ G}) \text{TrPI}_{\text{exp}}(\text{self}, E) \\ V ::= n \mid a \mid \text{self}() \mid [A_1 \mid A_2] \mid [A_1, \dots, A_n] \quad ; n \geq 0 \end{aligned} \quad \text{-(34A)}$$

Similarly, with guards, rule(26F) can be written as follows:

$$\begin{aligned} \text{TrPI}_{\text{match}}(\text{self}, \{P_1, \dots, P_n\}[\mathbf{when\ G}]\rightarrow E) &= \\ \left\{ \begin{array}{l} \text{self}(\dots, \text{input_pati}, \dots) \dots [\text{input_pati}=\text{TrPI}_{\text{arg}}(P_i)] \dots \mathbf{TrPI}_{\text{guard}}(\text{self}, \mathbf{when} \\ \mathbf{G}) \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Atoms, Numbers}\} \text{ or if } P_i \in \{\text{Variables}\} \text{ and name} \\ P_i \in \text{BNSRAV} \\ \text{self}(\dots, X, \dots) \dots \mathbf{TrPI}_{\text{guard}}(\text{self}, \mathbf{when\ G}) \text{TrPI}_{\text{exp}}(\text{self}, E) ; \text{ if } P_i \in \{\text{Variables}\}, \\ \text{where } P_i \text{ is a variable } X \text{ and name } X \notin \text{BNSRAV} \end{array} \right. \end{aligned} \quad \text{-(26G)}$$

7.3 IF Expression

The *if* expression has the following syntax:

if $G_1 \rightarrow E_1; \dots; G_n \rightarrow E_n$ end

The guards G_1, \dots are evaluated sequentially. If a guard succeeds then the related expression is evaluated. The result of the evaluation becomes the value of the *if* from. *If* guards have the same form as *function* guards. While translating such *if* expression in π -calculus, we have used non-determinism between the guard expressions as follows provided that all variables used in guards are bound:

$$\begin{aligned} \text{TrPI}_{\text{exp}}(\text{self}, \text{if } G_1 \rightarrow E_1; \dots; G_n \rightarrow E_n \text{end}) &:= \text{TrPI}_{\text{guard}}(\text{self}, G_1) \text{TrPI}_{\text{exp}}(\text{self}, E_1 + \dots \\ &\quad + \text{TrPI}_{\text{guard}}(\text{self}, G_n) \text{TrPI}_{\text{exp}}(\text{self}, E_n) \end{aligned} \quad \text{-(35)}$$

Chapter 8

Model Checking with HAL

In this chapter, HAL, an automata-based verification environment for the π -calculus is introduced. The π -models gained in the previous chapters can now be verified with HAL. The HAL system is able to interface with several model checking tools to determine whether or not certain properties hold for a given specification. However, in this chapter, we have only provided some LTS of some certain π -models. A detailed about working with HAL can be found in [9, 10].

	8.1 Introduction	-145
	8.2 HAL compatible π -calculus	-146
	8.3 HAL System Overview	-147
	8.4 HAL Commands	-147
	8.5 LTS from π -calculus Models	-148
	8.5.1 LTS of Program 5.1	-148
	8.5.2 LTS of Program 4.2	-149
	8.5.3 LTS of Program 5.3	-150
	8.5.4 LTS of Program 5.4	-150
Table of Contents \Rightarrow		

8.1 Introduction

History Dependent automata (HD-automata in short) have been proposed in [38, 39] as a new effective model for name passing calculi. Like ordinary automata, HD-automata are made out of states and labeled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation and name extrusion and we know these are the distinguished mechanisms of name passing calculi.

The HD-Automata Laboratory (HAL) is an integrated tool set for the specification, verification and analysis of concurrent systems. The HAL toolkit is the component of JACK[40] which provides facilities to deal with π -calculus by exploiting HD-automata. The goal of HAL is to verify properties of mobile systems specified in the π -calculus. Exploiting HAL facilities, π -calculus specifications are translated first into

HD-automata and then in ordinary automata. Hence, the JACK bisimulation checkers can be used to verify (strong and weak) bisimilarity. Automata minimization, according to weak bisimulation is also possible. HAL supports verification of logical formulae expressing properties of the behavior of π -calculus specifications. The ACTL[41] model checker provided by JACK can be used for verifying properties of π -calculus specifications, after that the π -logic formulae expressing the properties have been translated into ACTL formulae. The complexity of the model checking algorithm depends on the construction of the state space of the π -calculus agent to be verified, which is, in the worst case, exponential in the syntactic size of the agent.

8.2 HAL compatible π -calculus

The syntax of π -calculus agents supported by HAL is shown in Figure 8.1, where we use x to denote generic names and A to denote agent identifiers.

System:	$S ::= Q^+$	
Process Definition:	$Q ::= A(x_1, \dots, x_n) = \pi$ (where $i \neq j \Rightarrow x_i \neq x_j$) ; $n \geq 0$	
Process:	$\pi ::= \text{nil}$	Deadlock agent(Nil)
	$\alpha.\pi$	Prefix
	$\pi_1 \mid \pi_2$	Parallel composition
	$\mid (\pi_1, \dots, \pi_n)$	Parallel composition
	$\pi_1 + \pi_2$	Nondeterministic composition
	$+ (\pi_1, \dots, \pi_n)$	Nondeterministic composition
	$(x) \pi$	Restriction
	$[x=y]\pi$	Name matching
	$A(x_1, \dots, x_n)$	Agent identifier
	(π)	Parenthesis
Action Prefixes:	$\alpha ::= x?(y)$	Input
	$x!y$	Output
	tau	Silent

Figure 8.1 The π -calculus syntax compatible with HAL

Prefixes (input, output, tau, restriction and matching) take precedence on nondeterministic composition, which in turn takes precedence on parallel composition.

8.3 HAL System Overview

In the current implementation, the HAL environment consists essentially of five modules(cf. Figure 8.2): three modules perform the translations from π -calculus agents to HD-automata (pi-to-hd), from HD-automata to ordinary automata (hd-to-aut) and from π -logic formulae to ordinary ACTL formulae (pl-to-ACTL). The fourth module (hd reduce) provides routines that manipulate the HD automata. The fifth module is basically the JACK environment which works at the level of ordinary automata and performs the standard operations on them like behavioral verification and model checking.

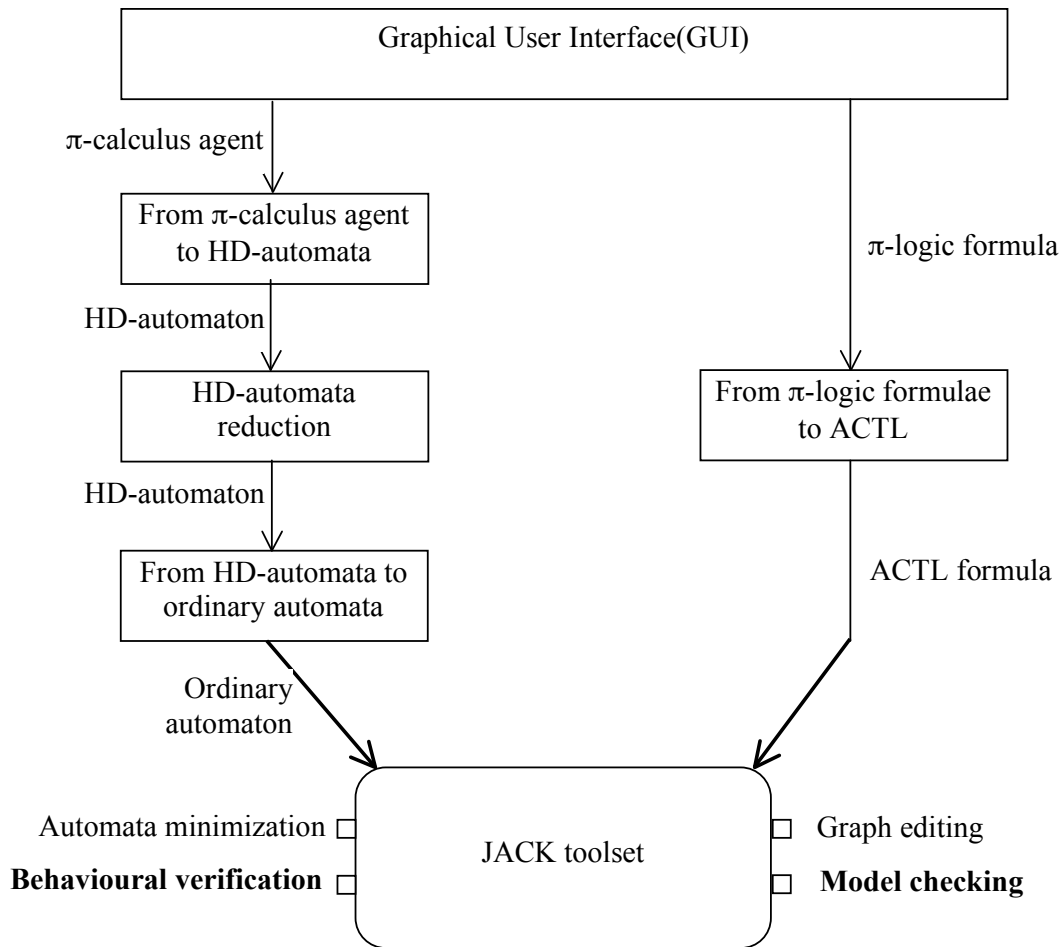


Figure 8.2 The logical architecture of HAL environment.

8.4 HAL Commands

HAL reads commands from the standard input. At the moment, it accepts the following commands:

define $A(x_1, \dots, x_n) = \pi$

This command defines a π -calculus agent. A is the identifier that is associated to the agent. x_1, \dots, x_n are the formal parameters and π is the body of the agent(cf. Figure 8.1).

build A

This command builds the HD-automaton for the agent corresponding to identifier A . The HD-automaton is saved in file $A.hd$.

const x

This command declares name x as a constant name. Constant names cannot be received by an agent as values of input transitions.

8.5 LTS from π -calculus Models

In this section, some π -models are used to get LTS from HAL.

8.5.1 LTS of Program 5.1

HAL compatible π -model of Program 5.1 is as follows:

```
define foo(self)=(send_res)(receive_res)(stop)(terminate)( self!stop.nil | send_res!
    stop.nil | send_res?(dummy).(self?(input_pat).[input_pat=stop]
    receive_res ! terminate . nil ))
```

```
define main()=(self)(foo(self))
```

```
build main
```

```
build foo
```

The LTSs of Program 5.1 are as follows:

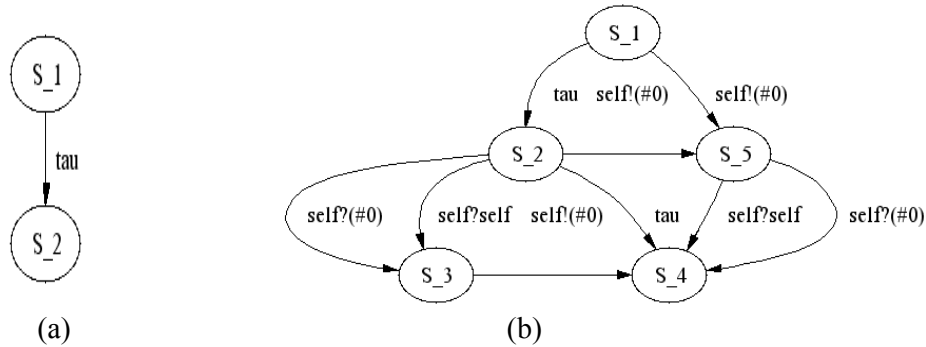


Figure 8.3 LTS of Program 5.1 (a) main process, (b) foo process

8.5.2 LTS of Program 4.2

HAL compatible π -model of Program 4.2 is as follows:

```

define s1(self)=(msg_a) (msg_c) (self ? (input_pat1).[input_pat1=msg_a]s2(self) +
    self ? (input_pat2).[input_pat2=msg_c]s3(self))

define s2(self)=(msg_x) (msg_h) (self ? (input_pat1).[input_pat1=msg_x]s3(self) +
    self ? (input_pat2).[input_pat2=msg_h]s4(self))

define s3(self)=(msg_b)(msg_y)(self ? (input_pat1).[input_pat1=msg_b]s1(self) +
    self ? (input_pat2).[input_pat2=msg_y]s2(self))

define s4(self)=(msg_i) ( self ? (input_pat).[input_pat=msg_i]s3(self))

define start(self)=(pid)(receiver_res) ( p) (send_res)(msg_a) ( p ! pid .nil | s1(pid) |
    receiver_res ? (dummy).nil | p ? (State_Pid). ( State_Pid ! msg_a
    .nil | send_res ! msg_a .nil ) )

define main()=(self) (start(sel.^`

```

```

build s1
build s2
build s3
build s4
build start
build main

```

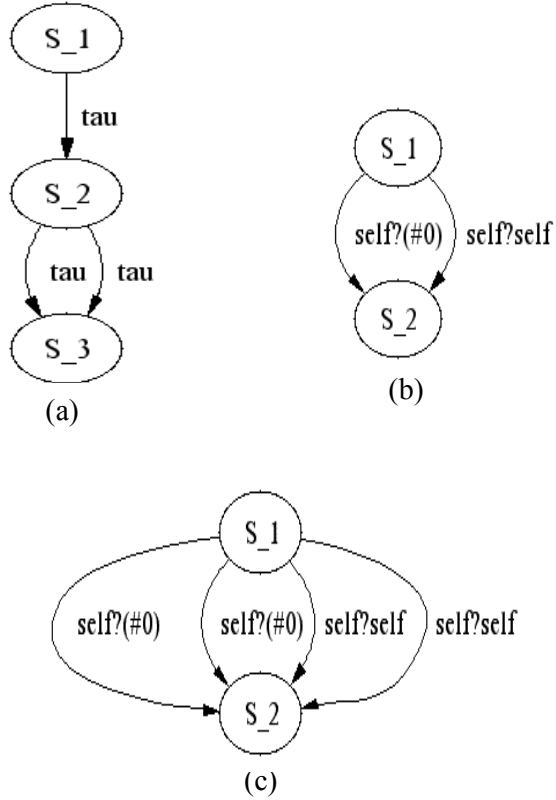


Figure 8.4 LTS of Program 4.2 (a) start process, (b) $s4$ process (c) $s1/s2/s3$ process

8.5.3 LTS of Program 5.3

HAL compatible π -model of Program 5.3 is as follows:

```
define ping(self)=(po_pid)(p)(pong_res)(pong_send_res)(ping_res)(pong)(ping) ( p !
    po_pid.nil | pong(po_pid)| pong_res ?(dummy).nil |?(Ping_ID).
    (Pong_ID!self. Pong_ID!ping.nil | pong_send_res ? (dummy).
    ( self ?(input_pat).[input_pat=pong]ping_res ! pong.nil ) ))
```

```
define pong(self)=(ping)(pong)(pong_res)(self?(Ping_ID).self?(input_pat).
    [input_pat=ping](Ping_ID!pong.nil | pong_res!pong.nil))
```

```
define main()=(self) (ping(self))
```

```
build ping
```

```
build pong
```

```
build main
```

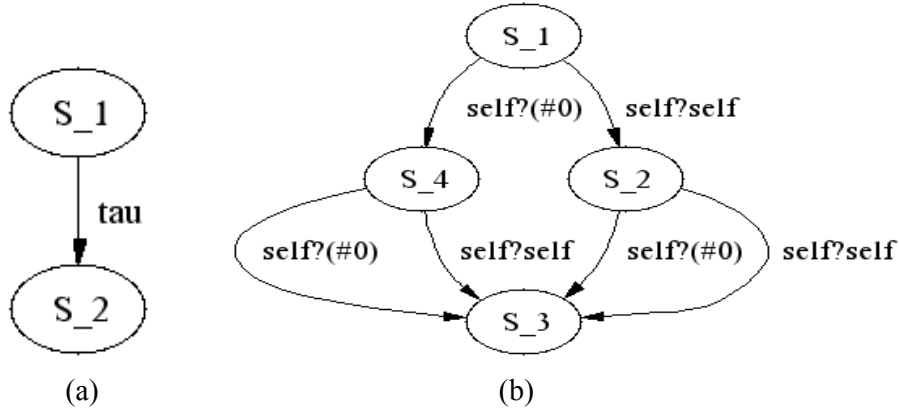


Figure 8.5 LTS of Program 5.3 (a) main/ping process, (b) pong process

8.5.4 LTS of Program 5.4

HAL compatible π -model of Program 5.4 is as follows:

```
define loop(self)=(loop_send_res)(Message)(self?(From).self?(Message).
    From!Message.nil | loop_send_res! Message.nil |
    loop_send_res?(dummy).loop(self))
```

```
define start(self)=(lpid)(p)(loop_res)(hello)(start_send_res)( p!lpid.nil | loop(lpid)|
    loop_res?(dummy).nil | p?(Loop_Pid).( Loop_Pid!self.
    Loop_Pid!hello.nil | start_send_res!self.start_send_res!hello.nil))
```

```
define main()=(self)(start(self))
```


build loop
 build start
 build main

With the recursive definition of **loop** process, it was not possible (there were infinite states) to get LTS from HAL, therefore, we have use nil process in place of recursive process **loop**. The new definition of **loop** process is as follows:

```
define loop(self)=(loop_send_res)(Message)(self?(From).self?(Message).
    From!Message.nil | loop_send_res! Message.nil |
    loop_send_res?(dummy).nil)
```

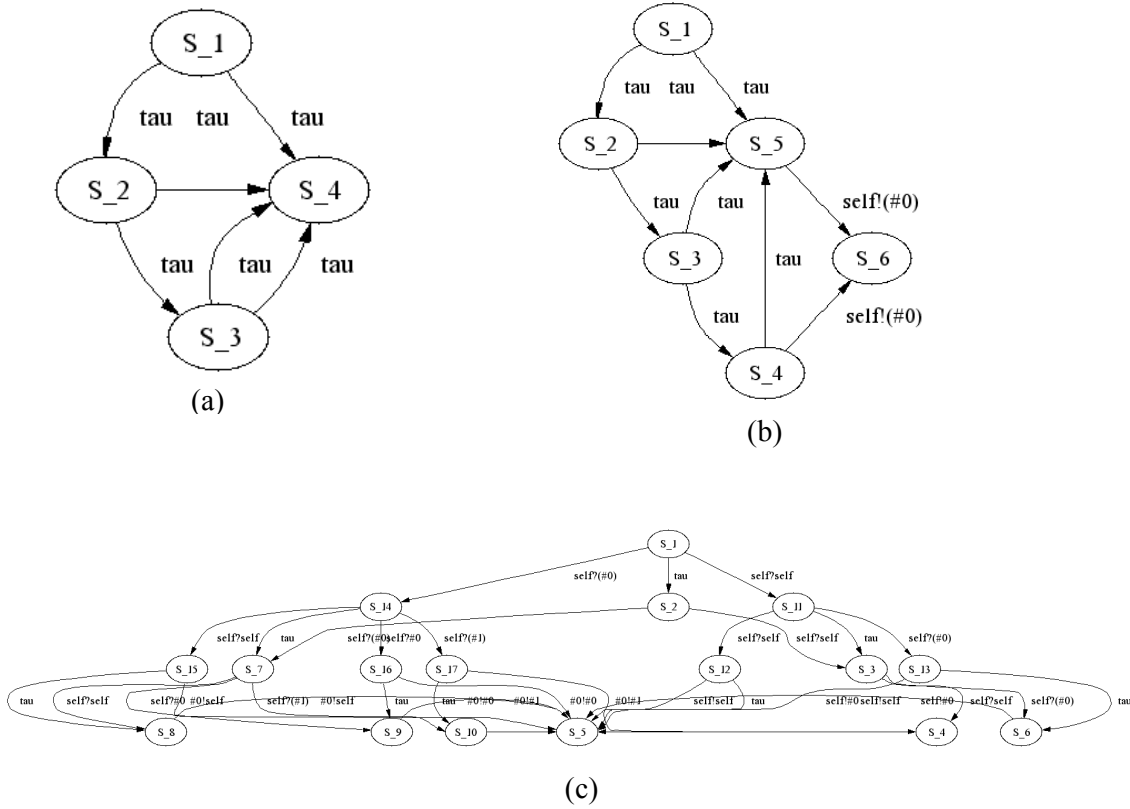


Figure 8.6 LTS of Program 5.4 (a) main process, (b) start process (c) loop process.

Chapter 9

Conclusion

In this thesis work, we have tried to translate the programming language Erlang into the π -calculus. We started from a restricted subset of Erlang and then gradually added more syntactic constructs for translation mapping. However, Erlang is a full programming language and it was beyond the scope of this thesis to consider all of its facets. For instance, in the case of *mailbox* of a process, we have not kept the full semantics (the ordering of messages in mailbox) of Erlang in the translated π -calculus system. In this section, we will point out some hints how this could be done along with some other hints of future works. Finally, we draw the conclusion with the summary of this thesis work.

Table of Contents \Rightarrow	9.1 Future Works	-152
	9.2 Summary	-154

9.1 Future Works

We have used the asynchronous π -calculus as the target specification language but one can model Erlang with the synchronous π -calculus too.

We have considered the *spawn/2* function with two arguments; first argument is the calling function name and second is the argument(s) of that calling function. One can consider other variants of spawn e.g. *spawn/3* and/or *spawn/4* for translation mapping.

As we have used the asynchronous π -calculus, ordering of messages in the *mailbox* of a process could not be respected. One could think of solving this problem. However, we have proposed one possible technique to solve such ordering problem by providing an abstract translation of such a *send/receive* subset of Erlang in asynchronous π -calculus.

As messages are sent asynchronously, their corresponding sub-processes are working in parallel with the receiving sup-process and thus, any of the sub-processes can communicate with the receiving sup-process, thereby violating the order of the messages in asynchronous π -calculus.

Message order is kept between two given processes in Erlang. To keep the message order, we can introduce a different mechanism to model the flow of messages. This

mechanism is the Erlang's *mailbox*. For each process *pid*, a *mailbox_{pid}* can be assumed according to Erlang semantics. A *mailbox* can keep any number of messages to one given process and it keeps the track of the order in which they arrive. To keep such semantics in translated π -calculus systems, one can develop a π -calculus implementation of the *mailbox* which behaves as follows:

$$\begin{aligned} & \text{TrPI}_{\text{exp}}(\text{self}, \text{pid} ! \mathbf{A}, \mathbf{E}) \parallel \text{mailbox}_{\text{pid}}(\mathbf{A}_1 \dots \mathbf{A}_n) \\ & (\text{react}) \\ & \Rightarrow \text{TrPI}_{\text{exp}}(\text{self}, \mathbf{E}) \parallel \text{mailbox}_{\text{pid}}(\mathbf{A}_1 \dots \mathbf{A}_n \mathbf{A}) \end{aligned} \quad -(9.1)$$

$$\begin{aligned} & \text{TrPI}_{\text{exp}}(\text{pid}, \text{X} = \text{receive } \mathbf{Y} \rightarrow \mathbf{Y} \text{ end}, \mathbf{E}) \parallel \text{mailbox}_{\text{pid}}(\mathbf{A}_1 \dots \mathbf{A}_n) \\ & (\text{react}) \\ & \Rightarrow \text{TrPI}_{\text{exp}}(\text{pid}, \mathbf{E}[\mathbf{X}/\mathbf{A}_1]) \parallel \text{mailbox}_{\text{pid}}(\mathbf{A}_2 \dots \mathbf{A}_n) \end{aligned} \quad -(9.2)$$

$$\begin{aligned} & \text{TrPI}_{\text{exp}}(\text{self}, \text{X} = \text{spawn}(\mathbf{f}, [\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n]), \mathbf{E}) \\ & = \text{new pid}(\text{TrPI}_{\text{exp}}(\text{self}, \mathbf{E}[\mathbf{X}/\text{pid}]) \parallel \text{TrPI}_{\text{exp}}(\text{pid}, \mathbf{f}(\mathbf{A}_1, \dots, \mathbf{A}_n)) \parallel \mathbf{mailbox}_{\text{pid}}()) \end{aligned} \quad -(9.3)$$

A sender process now transmits its message through the receiver's mailbox. Messages cannot switch order on the way. Of course, we need to spawn an inbox with each new process as shown in spawn in rule (9.3) above.

If the concepts of rules (9.1), (9.2) and (9.3) can be implemented in π -calculus, the ordering of messages could be kept from the sender's point of view. If there is only one match in receiving side, full semantics of *send/receive* mechanism of Erlang can be kept in π -calculus. But if there are more than one matches in the receiving side for the mailbox messages, we have used non-determinism between different matches with rule (14). Due to the non-deterministic choices between different matches, again there is a possibility of violating the order of the mailbox messages as any match could take part with the mailbox message. We have already provided an approach of solving this non-determinism in Section 4.9. Further improvements on this approach could be a good future work.

We have not provided the techniques for translation mapping *Lists* semantically. One can model such data structure in π -calculus using the similar approach mentioned here in the case of *mailbox*.

The *[after Time \rightarrow ActionTimeout]* of *receive* expression(cf. Section 2.6) can be modelled with real-time π -calculus as an interesting future work.

9.2 Summary

We have presented how a programming language supporting concurrent and distributed behaviors can be modelled to a specification language; from a complete program to a system model. We used Erlang as our programming language and the π -calculus as our target specification language and therefore, we have translated a program written in Erlang to a system model in π -calculus. By means of using several small examples, we have also shown that our translated π -models possess the same behaviors as it could be expected from their corresponding Erlang programs. Once the system model is achieved from Erlang program, it is possible to apply the model checking techniques for this system with existing tools, thus, an automatic verification of Erlang program is possible. The techniques we have applied here for translating an Erlang program to a π -calculus model can be applied for any programming language for gaining a model in the π -calculus. In some cases, where dynamic behavior of the program is not so important, CCS is a good choice as a target specification language and our developed methods can easily be reused or slightly modified to get a CCS system from such programming languages.

References

- [1] J. Armstrong, R. Virding, C. Wikström and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd Edition, 1996.
- [2] T. Noll. *Programming Concurrent Systems*. Lecture Notes, RWTH Aachen University, 2001.
- [3] T. Noll. *Formal Model of Concurrency*. Lecture Notes, RWTH Aachen University, 2003.
- [4] G. Boudol. *Asynchrony and the π -calculus*. Technical report, INRIA, 1992.
- [5] CADP: <http://www.inrialpes.fr/vasy/cadp/>
- [6] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–14, 2002.
- [7] Thomas Arts, Clara Benac Earle, and Juan José Sánchez Penas. Translating Erlang to μ CRL. *In Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004.
- [8] μ CRL: <http://homepages.cwi.nl/~mcrl/>.
- [9] HAL: <http://matrix.iei.pi.cnr.it/projects/JACK/hal.html>.
- [10] Gian-Luigi Ferrari, Stefania Gnesi, Ugo Montanari and Marco Pistore. A model-checking verification environment for mobile processes. In *Proc. TOSEM'03*, ACM Press, 2003, p: 440 – 473.
- [11] Christian Wiklander. *Verification of Erlang Programs Using SPIN*. Master's Thesis. Ericsson Computer Science Lab(CSLAB), 1999.
- [12] Sriram K. Rajamani and Jakob Rehof. Conformance Checking for Models of Asynchronous Message Passing Software. *Proceedings CAV 02, International Conference on Computer Aided Verification*.
- [13] Sagar Chaki, Sriram K. Rajamani and Jakob Rehof. Types as Models: Model Checking Message-Passing Programs *Microsoft Research Technical Report, MSR-TR-2001-71, August 2001*.

- [14] Sriram K. Rajamani and Jakob Rehof. A Behavioral Module System for the Pi-Calculus. *Proceedings SAS 01, Static Analysis Symposium, Paris, France, July 2001. Springer LNCS 2126*, 375-394.
- [15] Behave: <http://research.microsoft.com/behave/>.
- [16] P. Yang, C. R. Ramakrishnan, and S.A. Smolka, A Logical Encoding of the pi-Calculus: Model Checking Mobile Processes Using Tabled Resolution. *Proceedings of Fourth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, Lecture Notes in Computer Science, Springer-Verlag (Jan. 2003).
- [17] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-295, May, 1997.
- [18] Hosung Song and Kevin J. Compton. *Verifying pi -calculus Processes by Promela Translation*. Technical Report 2003. Department of Electrical Engineering and Computer Science University of Michigan, Ann Arbor, MI 48109, USA.
- [19] B. Victor and F. Moller. The Mobility Workbench — A tool for the π -calculus. In *Proc. CAV'94*, LNCS 818. Springer Verlag, 1994.
- [20] J.-C. Fernandez and L. Mounier. “On the fly” verification of behavioral equivalences and preorders. In *Proc. CAV'91*, LNCS 575. Springer Verlag, 1991.
- [21] MWB: <http://www.it.uu.se/research/group/mobility/mwb>.
- [22] B. Victor. *A Verification Tool for the Polyadic π -calculus*. DoCS Licentiate Thesis 94/50. Dept. of Computer Science, Uppsala University, May 1994.
- [23] M. Dam. Model checking mobile processes. In E. Best, editor, *CONCUR'93, 4th Intl. Conference on Concurrency Theory*, Vol. 715 of Lecture Notes in Computer Science, p. 22-36. Springer-Verlag, 1993. Full version in Research Report R94:01, Swedish Institute of Computer Science, Kista, Sweden.
- [24] CriSys: <http://www.cs.umn.edu/crisys/index.html>
- [25] Ola Samuelsson and Anders Frank *A Graphical User Interface for Erlang*. Uppsala University master's thesis, 1994.
- [26] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren,

- Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. *Core Erlang 1.0 language specification*. November 2000.
- [27] Maurice Castro. *Erlang in Real Time*. RMIT University, 2001.
 - [28] VeriSoft: <http://cm.bell-labs.com/who/god/verisoft/>
 - [29] K. Havelund, T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, STTT, 2(4) April 2000.
 - [30] SPIN. <http://spinroot.com/spin/whatispin.html>
 - [31] Open Source Erlang Distribution. Ericsson Software Technology AB, Erlang Systems, 1999. <http://www.erlang.org/>
 - [32] Jonas Barklund and Robert Virding. Erlang 4.7.3. *Reference Manual*, 1999.
 - [33] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π -calculus. In *Proc. CONCUR '96*, LNCS 1119, Springer Verlag.
 - [34] Davide Sangiorgi and David Walker. *The π -calculus. A Theory of Mobile Processes*. Cambridge University Press, New York, NY, 2001.
 - [35] Robin Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, New York, NY, 1999.
 - [36] Calculi for Mobile Processes. <http://lamp.epfl.ch/mobility/>.
 - [37] Promela. <http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html>.
 - [38] M. Pistore. *History Dependent Automata*. PhD. Thesis TD-5/99, Università di Pisa, Dipartimento di Informatica, 1999.
 - [39] U. Montanari and M. Pistore. Checking bisimilarity for unitary δ -calculus. In *Proc. CONCUR '95*, LNCS 962. Springer Verlag, 1995.
 - [40] JACK: <http://matrix.iei.pi.cnr.it/projects/JACK/>
 - [41] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Proc. Ecole de Printemps on Semantics of Concurrency*, LNCS 469. Springer Verlag, 1990.