IMPLEMENTING A COMPILER / INTERPRETER FOR π-CALCULUS

by

Christian Tabone

Supervisor: Dr. Adrian Francalanza

DEPARTMENT OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE

UNIVERSITY OF MALTA

May 2006

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.T. (Hons.)

To my loving parents who have always supported me
through the past years and have always been there for me.

Declaration

I, the undersigned, declare that the Final Year Project report submitted is my work, except where acknowledged and referenced.

Acknowledgements

I would like to express my appreciation to Adrian Francalanza, my supervisor, for his encouragement and exceptional guidance.

I would also like show my gratitude to my family and to all my friends, for their support throughout my studies.

Abstract

We consider π-Calculus as the foundation of our study, by analyzing the syntax and semantics that this notation offers. We then describe a simple typing convention that will be used to type-check π-Calculus programs. This is followed by the description of an intermediate representation of the π-Calculus, and we suggest how serval machine implementations can use this representation to simulate π-Calculus programs. A parser and compiler are constructed. These will translate π-Calculus program into this intermediate representation. Subsequently, we concentrate on David N. Turner's Abstract Machine, to develop a Stand-Alone Virtual Machine capable of interpreting π-Calculus, and simulating a correct execution on the semantic meaning of the given program. We tackle a number of optimizations that are incorporated with the architecture of the Stand-Alone machine, to produce a more efficient simulation. We present the architecture for an Interactive Virtual Machine to allow a user to communicate with the program during execution, and we give an illustration on the differences between a Stand-Alone virtual machine and an Interactive virtual machine. We then verify the correctness of the virtual machines' implementations, by presenting a number of examples. We conclude by examining the capability of the Interactive virtual machine, in creating an abstract layer between the implementation details of π-Calculus programs and the user. We illustrate how a typical user is unable to distinguish between two π-Calculus programs, that offer the same functionalities, but have a different internal implementation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The π-Calculus is part of the family of *Process Calculi*, which are all formal mathematical paradigms, used to model concurrent systems. The notation of Process Calculi describe a framework based on processes which execute concurrently, during which a pair of these processes can synchronize by communicating. The π-Calculus was developed with exclusive interest on the mobility of process communication, in concurrent systems. It was originally developed by Robin Milner, but many others contributed to this growing research. Throughout this dissertation we will be dealing with the *Polyadic π-Calculus*, which is a variant to the original π-Calculus.

The descriptive ability that π-Calculus offers, emerges from the concept of *naming*, where communication links, known as channels, are referenced using a naming convention. Hence, mobility arises by having processes communicating the channel names. This is a remarkable primitive perception, however, it gives the π-Calculus a practical expressive power, which, as a result, can provide building blocks for other complex concurrent languages. *"It would serve as a flexible intermediate language for compilers of concurrent languages"* - David N. Turner.

Our aim throughout this dissertation will be that of developing a machine capable of understanding π-Calculus notation, and which gives an interpretation to the meaning behind the given description. This leads us to the development of a compiler for the π-Calculus notation, which will 'understand' the notation and translate it to a more manageable form. Hence, an interpreting machine would apply rules to the π-Calculus notation, to give an interpretation.

1.1 Aims and Objectives

The following is a list of the main objectives we shall be trying to achieve through this study. The order in which these are given, reflects the order in which they should be carried out, since these goals build up on each other. Figure 1.1 gives a depictive outline of what we shall try to achieve, showing how the goals depend on each other.

- Learn what π-Calculus is about, comprehend its notation and variations to it, and be aware of the logical programming that can be constructed by π-Calculus.
- Reflect on other studies that have been completed, in order to picture where this dissertation stands next to these studies, and to gather ideas on what has already been achieved by others, and how this was achieved.
- Design a Parser and Compiler, which serves to translate π-Calculus programs into an intermediate code representation, where this representation is to be used by various Virtual Machines, each of which can be developed to accomplish different tasks.
- Develop a Virtual Machine capable of interpreting the intermediate code of π-Calculus programs, and perform an isolated execution of this code.

- Develop another Virtual Machine, which extends the functionality of the previous machine, by allowing external sources to interact during the execution of the program.

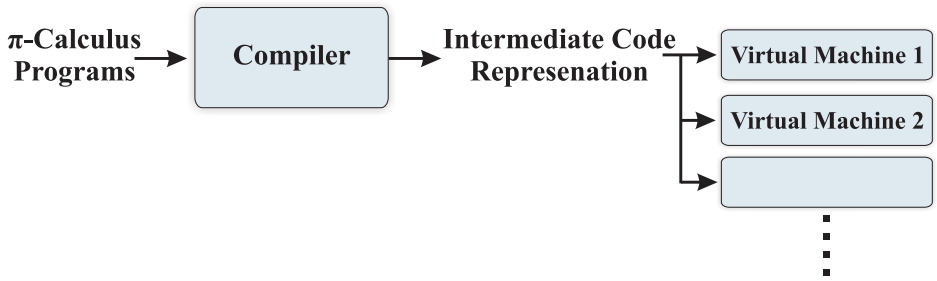


Figure 1.1: Overview of the objectives

1.2 Chapter Overview

Following this introductory chapter, the structure of this document almost follows the aims given in Section 1.1. Chapter 2 of this dissertation gives the background and the literature review on the critical points about the π-Calculus that are required to be understood for the rest of the study. We target the syntax and semantics of π-Calculus, we discuss a simple typing system and we look at related work.

In Chapter 3 we go on to the design of the translation modules, which involves the parsing of the π-Calculus source code, type-checking the program, and compiling it into an intermediate representation. We outline the components that make up the intermediate representation.

Chapter 4 gives the development of a *Stand-Alone Virtual Machine*, which is mostly based on Turner's Abstract Machine[Tur95]. We give a framework on which π-Calculus programs are correctly executed, followed by a number of optimizations to aid the performance of the Virtual Machine.

We then move on the Chapter 5, where we suggest the development of an *Interactive Virtual Machine*, and what benefits we acquire. We discuss a different design from that of the Stand-Alone Virtual Machine. However, we will give an analysis of the optimizations included in the Stand-Alone Virtual Machine and investigate if these optimizations will still be valid, when adopted by the Interactive Virtual Machine.

In Chapter 6 we give an evaluation analysis of the modules that have been developed throughout this dissertation. We give special interest to the Interactive Virtual Machine where we investigate this level of abstraction that this machine offers to the user. We develop two π-Calculus programs, with the same functionalities but different implementation details, and we discuss how the user is unable to distinguish one from the other.

We give our concluding remarks in Chapter 7, where we investigate what benefits have been acquired from this dissertation. We even give the limitation of the final application, and some ideas about the vision for possible future works.

Chapter 2

Background

This chapter consists of the literature review regarding the π-Calculus. We introduce *The Polyadic π-Calculus* by defining the Syntax Rules, Structural Rules and Reduction Rules, which we will use to build the compiler in Chapter 3 and the virtual machines in Chapters 4 and 5. We then give an overview of a simple typing system, such that we ensure that the communication between channels is done accurately. Finally we take a look at other studies, which are tightly related to this dissertation. These studies have developed various machines and architectures for π-Calculus, all of which, give us a motivation for the rest of this project.

2.1 The Polyadic π-Calculus

The π-Calculus notation models parallel processes, which can perform *input* or *output* actions through *channels*, thus allowing the processes to communicate. The message which is sent from one process to the other is a *name*, which gives a reference to a channel.

The communication of the channels' names themselves allow the processes to dynamically change the network of relations between them, through which communication is established.

Here we will be dealing with the *Polyadic π-Calculus*, which differs slightly from the original π-Calculus (*the Monadic π-Calculus*). The difference between the two is that, in Monadic π-Calculus, a single channel name is allowed to be exchanged during communication, while in the Polyadic π-Calculus, a list of channel names, known as a *tuple*, can be exchanged during a single interaction, where this *tuple* can possibly be empty.

In the Polyadic π-Calculus, a typing system for the channels is required to prevent *arity mismatch* between the input action and the output action. Arity mismatch happens when the number of sent channels does not match the number of arguments of the receiving action.

Interested readers should refer to the book "Communicating and Mobile Systems: The π-Calculus", by Robin Miler [Mil99]. In his book, Milner gathers most of his work, and introduces π-Calculus, as *"the new way of modeling communication"*. This book and other technical reports [Mil89a, Mil89b], again by Milner, were very useful during the study of this project.

2.1.1 Syntax

The syntax of the π-Calculus langauge that will be used throughout this dissertation is given in Table 2.1. In the π-Calculus the simplest form of entity is a *name*. A *name* is regarded as the referencing convention for a channel, through which processes can communicate.

$P, Q, R, S ::=$	$P \mid Q$	*Concurrency*
	$\mid$ if $x = y$ then P else Q	*Conditional Statement*
	$\mid$ $(\#z)P$	*Channel Scoping*
	$\mid$ $c![v_1, \ldots, v_n].P$	*Output Action*
	$\mid$ $c?(x_1, \ldots, x_n).P$	*Input Action*
	$\mid$ $* c?(x_1, \ldots, x_n).P$	*Replicated Input Action*
	$\mid$ $\tau.P$	*Internal Action*
	$\mid$ 0	*Null Process*

Table 2.1: Syntax for the π-Calculus

Example 2.1 (Using names): The action $c![a]$ in this system has the potential to output (send) the channel named a. This channel must be known since the action is trying to communicate it, hence it is using the channel name a.

$$c![a].Q$$

The π-Calculus notation makes use of channel variables. A variable represents a place-holder for a channel which is currently unknown, but has the potential to become known.

Example 2.2 (Using variables): In this example, the action $c?(x)$ has the potential to input (receive) a channel, and since it is not yet known which channel it will receive, then the variable x is used.

$$c?(x).P$$

The following is a description of the meanings behind the syntax given in Table 2.1. Further in-depth analysis can be found in [SW03a].

Concurrency means that process P runs independent of process Q. Both processes can communicate channel names between them by performing input/output actions on

a common channel. When multiple processes are running concurrently, we have non-deterministic communication.

A ***Conditional Statement*** will check for the equality of the two channels x and y, ($x = y$). If these channels are found to be equal, then the computation continues as P. Otherwise, if the channels are not equal, the computation continues as Q.

Channel Scoping[1] restricts the scope of the given name z, reducing its usage only to the process P. For instance, in example 2.3, the name z is private (or **bound**) to the system $z?(x).P$. Channel names which are not bound to any process are said to be **free names**.

Example 2.3 (Channel scoping): The process $z?(x).P$ is unable to interact with the process $z![a].Q$ through channel z, since these are actually different channels.

$$z![a].Q \mid (\#z)z?(x).P$$

An ***Output Action*** will communicate with another process, by sending the tuple of channel names $y_1, ...y_n$, through the channel c. When this is completed, the execution of the process continues as P.

An ***Input Action*** communicates with another process through channel c, where it receives a tuple of channel names. As we have already described, we have a tuple of variables, $y_1, ...y_n$, awaiting to receive the channel names from another process during communication. Thus these variables are substituted to channel names throughout the process, once these names are received.

This brings us to another channel scoping concept. Since occurrences of the variables in a process will be substituted by the channel names, these variables are bound to the process by the input action.

[1]Note that this is not the only operator for channel scoping, but this is indirectly obtained through the Input Action

Example 2.4 (Channel scoping by input actions): Note how in this system, the processes $c?(x).x![b].Q$ and $x?(y).R$ give the impression that these processes have potential to communicate through x. However, note how x is bound to the process, $c?(x).x![b].Q$, because of the preceding input.

$$c![a].P \mid c?(x).x![b].Q \mid x?(y).R \rightarrow P \mid a![b].Q \mid x?(y).R$$

A ***Replicated Input Action*** has the same behavior as an input action. The difference is that upon communication, it produces a copy of itself before proceeding with the execution, thus leaving an intact copy within the system.

Example 2.5 (Replicated input action): The action $*c?(x)$ releases a copy of the whole process that executes this action, upon communicating with another process.

$$c![a].Q \mid *c?(x).P \rightarrow *c?(x).P \mid Q \mid P$$

An ***Internal Action*** is regarded as an unobservable action which will simply continue as P, without affecting any surrounding processes.

A ***Null Process*** is an inactive process which cannot proceed to any further computations, and hence, it would simply terminate.

2.1.2 Reduction Semantics

The semantics of a language, is the meaning behind the expressions or the arrangement of terms, which are composed from the syntax of the language. In π-Calculus we deal with Reduction Semantics, which is the meaning behind the reduction of processes when these communicate. These semantics will be defined using a relation, $\rightarrow$, between two processes, showing how a process reduces its state, after that this has performed a single

s-assoc	$P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$
s-comm	$P \mid Q \equiv Q \mid P$
s-null	$P \mid 0 \equiv P$
s-scope	$(\#z)P \mid Q \equiv (\#z)(P \mid Q) \quad \text{if } z \notin fn(Q)$
s-scope-comm	$(\#z)(\#w)P \equiv (\#w)(\#z)P$
s-scope-null	$(\#z)0 \equiv 0$

Table 2.2: Structural equivalence rules for π-Calculus

action. For instance the process P is reduced to a different state as Q.

$$P \longrightarrow Q$$

We will use the closure of the relation, $\rightarrow$, to denote the reduction of a process P to Q in an arbitrary number of steps.

$$P \longrightarrow^* Q$$

We start familiarizing with the semantic meaning of our langauge by defining a set of axioms for the structure of the langauge. These *Structural Equivalence Rules* are given in Table 2.2. Such equivalences between different process states, permit us to manipulate the arrangement of the terms of a π-Calculus program, without affecting the semantics of the program. Hence, these equivalences give us control over the sequence of the terms, such that, Reduction Semantics can be applied to processes whenever applicable, regardless of the structure of the program.

s-assoc means that the concurrency operator, $|$, is associative between processes, meaning that the order in which parallel processes are executed does not affect the end result. Parentheses are used to denote operator precedence, however, it is important to note that the system will still execute non-deterministically.

Example 2.6 (Associativity): Note how the process, $c?(x).P$, can still reduce nondeter-

ministically with one of the other processes.

$$c![a].Q \mid \mathbf{(}c?(x).P \mid c!(b).R\mathbf{)} \equiv \mathbf{(}c![a].Q \mid c?(x).P\mathbf{)} \mid c!(b).R$$

s-comm denotes that, any two processes executing in parallel are commutative, meaning that the two can be interchanged between the concurrency operator, $|$, and still produce the same result.

Example 2.7 (Commutativity): Note how the position of the following two processes within this system, does not affect the communication.

$$c![a].Q \mid c?(x).P \equiv c![a].Q \mid c?(x).P$$

s-null implies that a process P executing in parallel with a null process, 0, is not affected with the presence of the null process, and neither is it affected if this null process is discarded.

s-scope implies that channel scoping applied on a process, P, can be applied to process Q, for a channel z, if the channel name z is part of the set of free names of Q. This means that the name z is not used within the process Q. Hence if its scope had to be expanded from P to $(P \mid Q)$, it will not affect the execution.

Example 2.8 (Structural equivalence in channel scoping): In the following system the scoping of the channel name z is expanded/reduced without affecting the end result, since the process $c?[x].0$ is unaware of the channel name z.

$$\mathbf{(\#z)(}z![a].0\mathbf{)} \mid c?[x].0 \equiv \mathbf{(\#z)(}z![a].0 \mid c?[x].0\mathbf{)}$$

s-scope-comm denotes that the sequence of two scoping operators is commutative, meaning that the two can be interchanged without changing the end result.

(r-communication)

$$c?(x_1, ..., x_n).P \mid c![y_1, ..., y_n].Q \rightarrow P\{^{y_1,...,y_n}/_{x_1,...,x_n}\} \mid Q$$

(r-replicated-communication)

$$*c?(x_1, ..., x_n).P \mid c![y_1, ..., y_n].Q \rightarrow *c?(x_1, ..., x_n).P \mid P\{^{y_1,...,y_n}/_{x_1,...,x_n}\} \mid Q$$

(r-tau)

$$\tau.P \longrightarrow P$$

(r-struct)

$$\frac{P \equiv P_1 \quad P_1 \longrightarrow Q_1 \quad Q_1 \equiv Q}{P \longrightarrow Q}$$

(r-concurrent)

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$$

(r-scope)

$$\frac{P \longrightarrow Q}{(\#z)P \longrightarrow (\#z)Q}$$

(r-condition-true)

$$\frac{x = y}{if\ x = y\ then\ P\ else\ Q \longrightarrow P}$$

(r-condition-false)

$$\frac{x \neq y}{if\ x = y\ then\ P\ else\ Q \longrightarrow Q}$$

Table 2.3: Reduction rules for the π-Calculus

s-scope-null means that a scoping operator does not have any effect on the null process, since this does not include any names.

To describe the reductions of the processes, we define a set of *Reduction Rules*, which formalize how processes communicate between themselves during their execution. Table 2.3 gives these Reduction Rules for the π-Calculus.

r-communication is the *communication rule* stating how two processes communicate through a common channel c. One of the processes has to **output** (send) a tuple of names through this channel , while the other process has to **input** (receive) these names, which are defined as a tuple of variables up to communication. Once that the names are received, the input variables are substituted by the received channel names throughout the process. We use the notation $P\{^{new}/_{old}\}$ to denote that the name old is substituted by the name new throughout process P.

Example 2.9 (Process communication): Consider the system of processes S. When these processes communicate, the tuple $[a, b]$ is sent from the process on the right side, to the process on the left side. This would result in having the variables $[x, y]$ replaced by the received values $[a, b]$ respectively throughout the process $x![c].P$.

$$\begin{aligned} S &\implies c?(x,y).x![c].P \mid c![a,b]Q \\ &\longrightarrow a![c].P\{^{a,b}/_{x,y}\} \mid Q \end{aligned}$$

r-replicated-communication is similar to **r-communication**, only that the replication action is performed. Upon communication, the process with the replicated input action will persist with a copy of itself.

Example 2.10 (Replicated communication): Consider the system S. When the processes communicate, the process $*c?(x, y).x![c].P$ is retained, and a new process is launched $c?(x, y).x![c].P$. This process will receive the tuple of channel names and continue reducing, as illustrated in example 2.9.

$$\begin{aligned} S &\implies *c?(x,y).x![c].P \mid c![a,b]Q \\ &\longrightarrow *c?(x,y).x![c].P \mid a![c].P\{^{a,b}/_{x,y}\} \mid Q \end{aligned}$$

r-tau gives the rule for *tau (τ)* reductions, or internal reductions, where an unobservable action is processed, reducing the process, independent of surrounding processes.

r-concurrent rule states that a reduction is possible for processes in a concurrent environment.

r-scope rule states that a reduction is possible for processes containing bound names, and the scoping of these channels remains unchanged.

r-struct infers that a reduction is possible from P to Q; if there is a reduction from P_1 to Q_1, and if P and Q are equivalent to P_1 and Q_1 respectively.

r-condition-true states that if x is found to be equal to y, then the conditional statement would reduce to the process P.

Example 2.11 (Conditional statement evaluating to True): Consider the system S, where the conditional statement is evaluated to **true**. Hence the process P follows the execution.

$$\begin{aligned} S &\Longrightarrow c![a,a].R \mid c?(x,y).\text{if } x = y \text{ then } P \text{ else } Q \\ &\longrightarrow R \mid P\{^{a,a}/_{x,y}\} \end{aligned}$$

r-condition-false states that if x is not equal to y, then the conditional statement would reduce to the process Q.

Example 2.12 (Conditional statement evaluating to False): Consider the system S, where the conditional statement is evaluated to **false**. Then the process Q follows the execution.

$$\begin{aligned} S &\Longrightarrow c![a,b].R \mid c?(x,y).\text{if } x = y \text{ then } P \text{ else } Q \\ &\longrightarrow R \mid Q\{^{a,b}/_{x,y}\} \end{aligned}$$

So far, we have only witnessed examples which illustrate the π-Calculus notation and sematic rules. The following example offers a more useful concept that π-Calculus can be used for.

Example 2.13 (Memory cells): Consider a process using the channel named $cell$ to output a value that have been received earlier in the execution. For instance,

$$cell?[value].cell![value].P$$

Note how, this process is temporary storing the name received as the variable $value$.

Now, let us used the replicated input action to allow us to get hold of a cell for an unlimited number of times, and use the scoping operator to give each of the created cells a unique name. For example,

$$*createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0)$$

Observe how this process would communicate with another process, in the following system.

$$\begin{aligned} & *createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0) \\ \mid\ & createcell![helloworld, listener].listener?(myfirstcell).myfirstcell?(message).Q \end{aligned}$$

The replicated process communicates with our process through the channel $createcell$. Our process passes the name $helloworld$ that will be stored, and the name listener. The channel listener will thus communicate back the cell holding the channel helloworld.

$$\begin{aligned} & *createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0) \\ \mid\ & (\#cell)(cell![value].0 \mid getcell![cell].0)\{^{helloworld,\ listener}/_{value,\ getcell}\} \\ \mid\ & listener?(myfirstcell).myfirstcell?(message).Q \end{aligned}$$

What follows is the scoping operator that executes on the channel cell. Since the scoping operator restrict the channel $cell$, by creating a unique channel $cell_0$ and substituting $cell$ by $cell_0$ throughout the concerned processes. Hence, we will end up with a unique 'cell' waiting to output the name $helloworld$ and our channel $listener$ waiting to output the name just created channel $cell_0$.

$$\begin{aligned} & *createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0) \\ \mid\ & cell_0![helloworld].0\{^{cell_0}/_{cell}\} \\ \mid\ & listener![cell_0].0\{^{cell_0}/_{cell}\} \\ \mid\ & listener?(myfirstcell).myfirstcell?(message).Q \end{aligned}$$

Next our process will communicate through the channel $listener$, to input the channel representing the cell.

$$\begin{aligned} & *createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0) \\ \mid\ & cell_0![helloworld].0 \\ \mid\ & cell_0?(message).P\{^{cell_0}/_{myfirstcell}\} \end{aligned}$$

Finally our process will retrieve back the name $helloworld$ that was stored earlier.

$$\begin{aligned} & *createcell?(value, getcell).(\#cell)(cell![value].0 \mid getcell![cell].0) \\ \mid\ & P\{^{helloworld}/_{message}\} \end{aligned}$$

2.1.3 Simplifications and Assumptions

In the syntax that we defined, two key simplifications were made to the original Polyadic π-Calculus. We omitted completely the *Summation* operator, and we restricted *Replication* to Input actions only. This will simplify the virtual machines' development and their performance drastically.

The Summation operator gives a choice for communication to multiple processes, where these processes cannot communicate among themselves. For instance, in the example below processes P, Q and R cannot communicate among themselves, and only one of these can perform communication with process M - this is the choice. When communication occurs, the chosen process follows the execution, while the others are discarded.

$$(P + Q + R) \mid M$$

The main reason for removing the summation operator, is because it requires huge amounts of memory usage when compared to the other π-Calculus operators, and it is not very use-

ful and is rarely used. Turner in his thesis [Tur95], states how the experiment with PICT [PT00] has shown this.[2]

The second simplification that we undertook (discussed in detail by Turner [Tur95]), is the restriction of replication to processes with input actions, which we called Replicated Input. Full replication of a process is when a process copies itself upon performing an action, regardless of what that action is. The structural equivalence that was omitted, which defined replication is given below.

$$*P \equiv P \mid *P$$

If we had to allow this axiom to be observed, then there will be no control over how this operator would replicate the processes. This would result in having the process replicate itself to infinity.

$$*P \longrightarrow P \mid P \mid P \mid \ldots \mid P \mid *P$$

By restricting replication to input action, we avoid having this situation, since a process will only be replicated when this communicates through a replicated input action. In addition to this, full replication can be easily encoded by just using replicated input.

$$*P = (\#c)(*c?(\,).(P \mid c![\,]) \mid c![\,])$$

2.2 Channel Typing

In this section we define a simple typing system, so that we will be able to control *Arity Mismatch* during communication. We shall not go into great detail since this is beyond the scope of this work, however one can find further explanations in [SW03b, Kob]. We will use typing in both of the virtual machines that we shall develop throughout this project.

[2] Please note that minor choices can be encoded using the conditional statement instead of the summation operator.

Specific details on how typing is used within the various virtual machines, are given in the forthcoming chapters.

Arity mismatch is triggered when a process P communicates with another process Q, and the number of channel names that process P is outputting differs from the number expected by process Q while inputting. This will result in a runtime failure, which is highly undesirable. Examples 2.14 and 2.15 presents simple situations where arity mismatch will result in a runtime error.

Example 2.14 (Arity Mismatch): Consider the following two processes running within a system.

$$c![b_1, b_2].P \mid c?(x_1, x_2, x_3).Q$$

The process $c![b_1, b_2].P$ is unaware of the number of variables the other process has. Thus communication will commence. However, after that the second name is sent and received, process $c![b_1, b_2].P$ continues as P, but process $c?(x_1, x_2, x_3).Q$ is still waiting for an input on the variable x_3. This input will never happen. Hence this process locks up and it cannot proceed with its execution. It is this situation that triggers a runtime error.

Example 2.15 (Arity Mismatch): Now consider the following system of processes.

$$c![b].b![a].P \mid c?(x).x?(y_1, y_2).Q$$

At a first glance, these two processes look promising, and it seems that both will communicate through channel c, and continue running without triggering any runtime errors. However, after these perform the communication, the situation changes completely, since the process $c?(x).x?(y_1, y_2).Q$ has received the channel name b, and once the variable x is substituted by this channel name, we get the following situation,

$$b![a].P \mid b?(y_1, y_2).Q\{^b/_x\}$$

Note how now, the system will perform a runtime failure, as explained in example 2.14.

(err-communication) $\dfrac{}{a![V].P \mid a?(X).Q \longrightarrow_{\text{ERROR}}}$ if $V \neq X$	**(err-tau)** $\dfrac{P \longrightarrow_{\text{ERROR}}}{\tau.P \longrightarrow_{\text{ERROR}}}$
(err-scope) $\dfrac{P \longrightarrow_{\text{ERROR}}}{(\#z)P \longrightarrow_{\text{ERROR}}}$	**(err-concurrent)** $\dfrac{P \longrightarrow_{\text{ERROR}} \ or \ Q \longrightarrow_{\text{ERROR}}}{P \mid Q \longrightarrow_{\text{ERROR}}}$
(err-struct) $\dfrac{P \equiv P_1 \ \ P_1 \longrightarrow_{\text{ERROR}}}{P \longrightarrow_{\text{ERROR}}}$	**(err-condition)** $\dfrac{P \longrightarrow_{\text{ERROR}} \ or \ Q \longrightarrow_{\text{ERROR}}}{\text{if } x = y \text{ then } P \text{ else } Q \longrightarrow_{\text{ERROR}}}$

Table 2.4: Runtime errors for π-Calculus

We define the formal notation for when a system will produce a runtime error in Table 2.4. This table gives all the possible states of a how a system handles a runtime error. Most important is to note the rule **err-communication**, which triggers the error. Note the error's trail through the different states of the system.

To avoid our machines running into such a runtime error, we attach a *type* to the channels used within a π-Calculus program. The syntax for the typing of channels is given in Table 2.5. A channel c is attached to a type T, such that, this type T gives a mould for the tuple of names, that can be communicated through the channel c. By using types, before execution starts, the number of names that can be communicated (input/output) is retrieved from the type. If this number differs from the number of names (for outputs), or the number of variables (for inputs), execution is withheld.

As one can note, the structure of a Channel Type definition T, is a tree like structure, where the nodes represent a channel, and its children represent the tuple that is allowed through the channel. Therefore, the number of children nodes of a type, symbolizes the number of names allowed through the channel. This means that the leave nodes symbolize an empty tuple or list of names.

We are also defining *Type Variables*, denoted by V, which can be declared just like chan-

c, b, a, T	$::=$	$\langle\, T,\, T,\, \ldots,\, T\,\rangle \mid V$	*Channel Type*
V	$::=$	$\langle\, T,\, T,\, \ldots,\, T\,\rangle \mid \mu_X V$	*Type Variable*
$\mu_X V$	$::=$	$\langle\, {}_X V\,\rangle$	*Recursive Types*

Table 2.5: Typing syntax for π-Calculus

nel types, which can then be assigned to channels. This simplifies the notation for complex types, where the tree symbolizing these types is of a considerable depth.

Example 2.16 (Using channel types and type variables): The following is a declaration for the type of the channel c, which is using the variable type V.

$$c ::= \langle\, \langle\,\rangle\,,\, V \rangle$$
$$V ::= \langle\, \langle\,\rangle\,,\, \langle\,\rangle\,\rangle$$

A quick analysis shows that the channel c is being used correctly in the forthcoming process. Note how the variables x and y are not declared, but these use the types which are symbolized as the children of the declared channel c.

$$c?(x, y).x![\,].y![x, x].Q$$

Note that c could have been declared as

$$c ::= \langle\, \langle\,\rangle\,,\, \langle\, \langle\,\rangle\,,\, \langle\,\rangle\,\rangle\,\rangle$$

Recursive types are more complex then the 'simple' types, however, these allow us to produce more useful π-Calculus programs. This is why recursive typing is included in this study, but it is important to note, that this is a variant to proper recursive typing which is beyond our target. A channel declared as a recursive type must be done through a type variable, where this type must contain the same channel type within the declaration. The notation $\mu_X V$ means that the variable X must be declared within the declaration.

Example 2.17 (Recursive Typing): Here, the channel c is assigned to the type variable X. This type variable X is then defined as a recursive type with a single child, where this

child is again the type variable X.

$$c ::= X$$
$$\mu X ::= \langle X \rangle$$

For instance the following processes are correctly typed, according to the declarations that were made earlier. This example illustrates how the name for channel c is communicated through channel c itself.

$$c![c].P \mid c?(x).x![c].Q$$

To control these type declarations, we assign a set of environments to each process, which will store all the type declarations that are required throughout the process. We denote the set of type environments, which are mappings from channel names to types, of a process P by

$$\Gamma \vdash P$$

To add a mapping from a channel name c to its declared type T, to the set of type environments, we use the notation,

$$\Gamma_{c \mapsto T} \vdash P$$

To retrieve the type T of the channel c, from the set of type environments we use the notation,

$$\Gamma_{(c)} \vdash c ::= T, P$$

Finally, to check the equality of types we use the notation,

$$\Gamma \vdash \text{if } T = V,\ P$$

Table 2.6 gives the formal notation of the Type-Checking rules that should be followed

(tc-output)

$$\frac{\Gamma_{(a)} \vdash a ::= \langle\, T \,\rangle,\ P \quad \Gamma \vdash \text{if } V = T,\ P \quad \Gamma_{V \mapsto T} \vdash P}{\Gamma \vdash a![V].P}$$

(tc-input)

$$\frac{\Gamma_{(a)} \vdash a ::= \langle\, T \,\rangle,\ P \quad \Gamma_{X \mapsto T} \vdash P}{\Gamma \vdash a?(X).P}$$

(tc-replicated-input)

$$\frac{\Gamma_{(a)} \vdash a ::= \langle\, T \,\rangle,\ P \quad \Gamma_{X \mapsto T} \vdash P}{\Gamma \vdash *a?(X).P}$$

(tc-concurrent)

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$$

(tc-scope)

$$\frac{\Gamma_V \vdash P}{\Gamma \vdash (\#V)P}$$

(tc-tau)

$$\frac{\Gamma \vdash P}{\Gamma \vdash \tau.P}$$

(tc-null)

$$\frac{}{\Gamma \vdash 0}$$

(tc-condition)

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \text{if } x = y \text{ then } P \text{ else } Q}$$

Table 2.6: Type-checking rules for π-Calculus

before the start of a program execution. Note that these rules are observed on the static notation of the π-Calculus program, before any Reduction Semantics are undertaken.

The most vital rules are **tc-output**, **tc-input** and **tc-replicated-input**. These rules show how output actions check for type equality, before the rest of the process is type-checked. On the other hand input actions do not check for type equality, because these use variables, but add the variable value to the set of type environments to be type-checked later, if this is used by an output action.

2.3 Related Studies

In the following section we take a look at some of the most important and influential works, which contributed to the field of π-Calculus. These technical reports were useful

during the research of this study. Hence, this section will help the reader to understand and grasp a more solid idea on the π-Calculus and its uses. This section should give the reader a clear picture of where this dissertation stands, in comparison to other studies.

We want to point out that the work in this dissertation is not completely original, but rather ideas were adopted from the reports in the coming section. Please note that most of the work achieved by Turner, Sewell and Wischik is beyond the scope of our work here. Thus, the discussion targets only the objectives, and gives briefings to the ideas these people developed, and no complex explanations will be tackled.

2.3.1 A PICT-to-C Compiler

In his Ph.D thesis [Tur95], David N. Turner aims at developing a PICT-to-C Compiler. PICT [PT00] is a concurrent programming language built on the π-Calculus constructs, and was developed by Benjamin C. Pierce and David N. Turner himself. Turner investigates the π-Calculus by examining whether it is able to have a strongly-typed concurrent programming language based on the foundations of the π-Calculus. This is much beyond our target, but by reading through Turner's work we capture important explanations on the π-Calculus and on *Channel Typing*. In his chapter *'An Abstract Machine for π-Calculus'*, he describes useful applications which are suitable for our implementation. What really interests us, is how Turner considers implementing a π-Calculus machine, and making it as efficient as possible by introducing certain optimizations.

Turner builds his abstract machine on a uniprocessor platform. This means that, the processes are actually being simulated to work in parallel, since the underlying machine is utilizing a single processor. Turner states that a distributed implementation poses much further challenges. (This kind of implementation was later handled by Sewell, and later again by Wischik).

It is important to mention that David Turner eliminates the *Summation* operator in his description of the abstract machine. He suggests that the Summation operator affects the implementation by imposing complex issues, and that this is not feasible, since it is rarely used within a system.

Another essential point to note here, is how Turner tackles a **fair execution** of π-Calculus, which is being handled on a uniprocessor. Take into account that a single processor imposes a deterministic environment when the π-Calculus is a non-deterministic modal.

After facing and discussing these design issues, Turner's objective turns to refine the abstract machine, by making critical changes, in order to allow a more efficient execution. One of these optimizations, which we will be considering in our implementation, is the usage of *Environments* instead of substitution. Substitution is considered to be highly cost effective in terms of processing power, but this will be discussed in detail later.

2.3.2 Nomadic-PICT

Peter Sewell, in contrast to Turner, tries to solve the problem of having a π-Calculus machine, which will handle distributed programming, which works for a real distributed environment. In his works, Sewell tries to advance on Turner's work concerning PICT, from a simulation, to the actual concurrent infrastructure. Sewell handles this by introducing what he calls an "Agent Programming Language"; the *Nomadic-PICT*. The Nomadic-PICT is actually an extension to Pierce's and Turner's PICT.

Solid ideas about the background of Sewell's works is given by Sewell himself in a tutorial [Sew00], where he starts by introducing how π-Calculus can be applied to a distributed system. He uses the PICT programming and typing fashion to discuss certain issues.

In [PSP98, PSP99], Sewell, together with Wojciechowski and Pierce, proposes a simple calculus for mobile agents, such that these will be able to be transported between different locations within a distributed system, and regardless of this, they will still be able to communicate by message passing.

The description of the Nomadic-PICT implementation is given in [SW], where Sewell states that to have a high level programming framework for such distributed systems, the current infrastructure will face three main problems. The first problem is that programming distributed algorithms are very complex and fragile. The second problem is that the underlying structure of the present distributed systems, is not flexible enough to support a clean execution of distributed programming using π-Calculus. Finally, the underlying infrastructure of distributed systems is not application specific as it is required.

These problems are addressed directly in [PSP98, PSP99], where a small calculus is used and is closely related to real network communication protocols. This work then evolves to the *Two Level Architecture* that is adopted in implementing the Nomadic-PICT. In brief, the idea is to have the top level working closely to the programmer, where development is abstracted away from the lower levels of distributed systems. Hence, the second level is used to translate the program into a lower programming framework specifically to the underlying distributed system. This lower level will work closely to the present networking infrastructure and communication protocols.

2.3.3 The Fusion Machine

Lucian Wischik introduces *Explicit Fusion Calculus* in his Ph.D. dissertation [Wis01]. Explicit Fusion Calculus is an extension to Robin Milner's π-Calculus. This variant defines the details of how two processes communicate and exchange names. Lucian Wischik explains this calculus by segmenting the communication procedure into three main steps.

First, when two process are aware that the two can synchronize, these become "*fused*", meaning that both processes are bound to each other until they complete the communication. Next, these two processes assume that the variable of the input action is equal to the name being sent by the output action. Finally, the variable is discarded, resulting in having a "clean" substitution.

Wischik uses this Explicit Fusion Calculus to implement a concurrent and distributed machine for π-Calculus. Wischik's work differs from that of Sewell, since the development is actually that of an abstract machine based on π-Calculus, which will work at a low level of abstraction and on a real distributed system. On the other hand, Sewell achieves a distributed environment, by extending Turner's work, by applying an extra upper layer to operate on top of the already constructed abstract machine.

Wischik calls his abstract machine *The Fusion Machine*. He manages to accomplish his goals by using a technique which he names *Fragmentation*, where π-Calculus programs are fragmented into multiple simpler programs, in order to make them more transportable. These programs are fragmented using theories found in the Explicit Fusion Calculus. After, fragmenting the programs accordingly, at specific actions, these new programs can be distributed over multiple machines. Therefore, these programs can execute concurrently over a distributed system, and would communicate to each other by using Explicit Fusion Calculus. For communication, a common shared memory location is used. This acts as the name repository during Explicit fusion.

This is a brief idea on what Lucian Wischik managed to achieve. Its purpose is to motivate attracted readers, however at this stage this is beyond the scope of this project.

Chapter 3

Compiler Design

In this chapter we discuss the design and the implementation of a compiler. The purpose of the compiler is to translate the syntax of a π-Calculus program into an intermediate code representation, which can then be executed on several Virtual Machines. In Chapters 4 and 5 we develop these virtual machines, on which we can simulate this intermediate representation.

We shall be dealing with the highlighted section given in Figure 3.1. The development of a compiler requires other elements and modules to be developed before the actual compiler. [App02] offers indepth detail regarding compiling theories. We will start by constructing a π-Calculus language based on the given syntax in Table 2.1. We generate a Parser using a compiler generating tool, which will check a given program for syntactical errors, and output an Abstract Syntax Tree of the program. We then develop a Type-Checker module, which implements the rules given in Table 2.6, on the Abstract Syntax Tree. Finally we design a compiler that will convert an Abstract Syntax Tree (Type-checked or not), into an intermediate code representation.

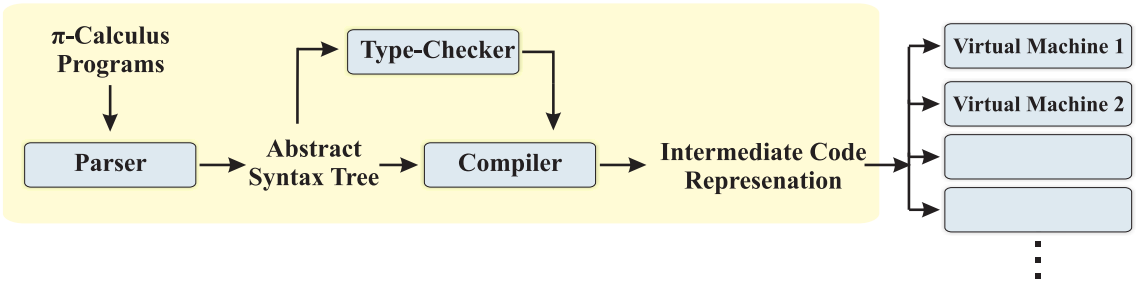


Figure 3.1: Compiler Architecture

3.1 The Π-Language

The language that is constructed is an extension to the syntax given in Table 2.1. For reference and explanation purposes, we shall call this language Π-Language. The extensions that were introduced to the Π-Language are mostly programming constructs making the π-Calculus more programmable. Figure 3.2 shows a typical π-Calculus program as it would be constructed using the Π-Language. The implementation details found within the various sections of a Π program, are written using the syntax given in Table 2.1, which has already been explained. Hence, in this section, we will discuss the constructs that have been added. The grammar describing our language is given in EBNF format (Extended Backus-Naur Form), and can be found in Appendix B

3.1.1 The Include Section

As an option, the programmer can use the `include` keyword to import other compiled π-Calculus files. Compiled files are π-Calculus programs, written in the Π-Language, which have been processed by the compiler, and saved as intermediate code. For example, in Figure 3.2, both *send* and *receive* are compiled programs, which are being imported, and incorporated with this particular program. These programs will not be compiled again here, but the intermediate code is simply added to the rest of the program. Such files can

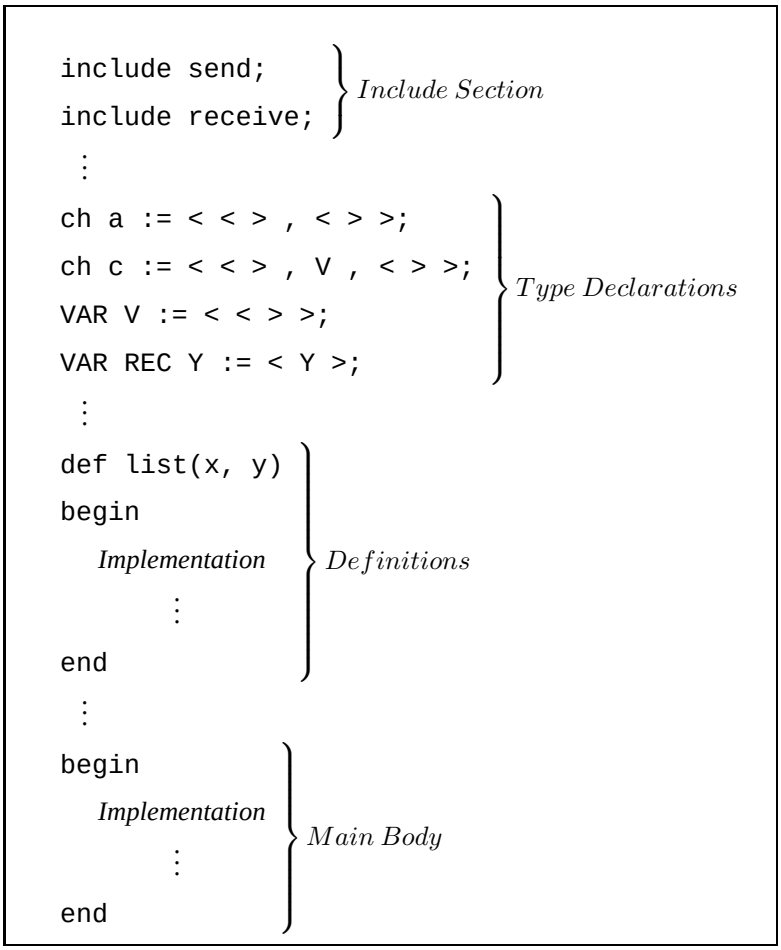


Figure 3.2: A typical π-Calculus program written in Π-Language

be used as libraries for π-Calculus definitions as we shall see later.

3.1.2 Type Declarations

Channel types are declared in this section, and the syntax used is very similar to the typing syntax given in Table 2.5. The same principles explained in Section 2.2 apply for the Π-Language. Note that the convention for channel name is to use small caps while, variable names are given in capitals.

1. **Channel types** The syntax for channel types is as follows:

   ch *name* **:=** *structure/variable* **;**

   The keyword `ch` declares the a channel and is given the name *name*. The assignment `:=` symbol, allocates the *structure* or *variable* to the channel. The semicolon signifies the termination of the declaration. The structure for the declaration follows the same principle of the typing syntax in Table 2.5. For instance in Figure 3.2, we are declaring two channels, *a* and *c*.

2. **Type Variables** are very similar to Channel type declarations. The difference is that the keyword `var` is used to declare a variable, and the name of the variable has to be given in capitals. The semicolon signifies the termination of the declaration.

   var *NAME* **:=** *structure/variable* **;**

   In our example given in Figure 3.2, a type variable *V* is being declared. It is critical to understand that type variables are only used for other type declarations and cannot be used outside the declarations section.

3. **Recursive Types** are declared means of a recursive variable. The procedure is to first declare the recursive type as a variable, then use this variable to declare a

channel. This gives us more control over recursive typing. The syntax for recursive types is as follows

var rec *NAME* **:=** *structure*$_{(NAME)}$ **;**

These types are similar to type variables, only that the keyword `rec` is used after the `var` keyword to indicate the presence of recursion. The structure for this type has to include the name of the type itself. The semicolon signifies the termination of the declaration. In Figure 3.2 the recursive variable *Y* is being declared.

3.1.3 π-Calculus Definitions

A definition is like a program function, where a piece of π-Calculus program can be modularized, in order to be reused whenever it is required. For a programmer to define a definition, the `def` keyword has to be used, followed by a number of variable parameters enclosed in parentheses, like the example given in Figure 3.2.

The implementation section within the definitions is constructed using the π-Calculus syntax given in Table 2.1. This implementation must be enclosed in between `begin` and `end` keywords. Any number of definitions can be constructed within a single program.

A definition can be called by simply calling the given name, and passing the required channel names as the arguments. For instance, to call the definition in Figure 3.2,

```
list(a,b).P
```

3.1.4 The Main Body

The main body of a program is the starting point of the execution. In this section the programmer will construct π-Calculus code using the syntax in Table 2.1. The main body is an optional section, so that the programmer has the choice of omitting the main body and implementing only a number of definitions, hence composing a library of definitions. The implementation of the main body has to be enclosed in between the `begin` and the `end` keywords.

The main body should be the last piece of code that is defined, meaning that any type declarations and π-Calculus definitions should precede this section.

3.2 The Parser

The grammar specification given in Appendix B, for the Π-Language, was used to generate parser modules by using the *JavaCC* tools. JavaCC (Java Compiler Compiler) provides a set of tools (JavaCC, JJTree and JJDoc), which are able to understand a scripting language based on ENBF, and generates parser and compiler classes for the language that was specified in the EBNF specifications. These tools construct essential modules, for the Parser section, and interface modules on which the Type-Checker and the Compiler can be implemented.

At this stage, it is important to understand the task of the Parser, even though this will be generated by the JavaCC tools. This is necessary, since the development of the compiler will utilize many of the Parser features.

The Parser's job is to scan through a stream of characters, where the sequence of these

characters make up the current π-Calculus program, written in the Π-Language defined in Appendix B. This raw text is fed to a component known as the *Lexer* or the *Tokenizer*, where **Lexical Analysis** is performed. This filters out the code by removing unwanted text, such as comments and trailing white characters (spaces, carriage returns, tabs, etc.). The Tokenizer outputs a stream of tokens, where each token represents an entity of the langauge. For example, the keyword `begin`, the channel c and the action $?$, would all be tokens.

These tokens are then passed to the main Parser module, where its job is to check that the sequence of the tokens matches the grammar, (as defined in Appendix B), or more specifically it checks for syntax errors. The Parser will output a data structure known as an *Abstract Syntax Tree*, which is built from the tokens depicting the structure of the program. The Syntax Tree gives the program its structure and a more sematic meaning.

Example 3.1 Parsing a π-Calculus implementation section: For the following π-Calculus implementation, the parser will generate the Abstract Syntax Tree in Figure 3.3.

$$c![a].0 \mid c?(x).0$$

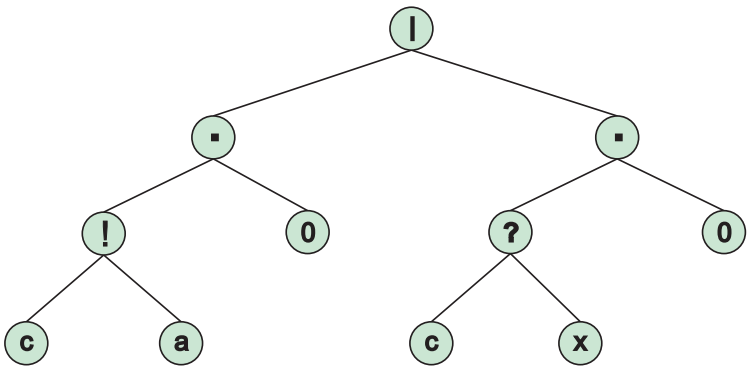


Figure 3.3: Abstract Syntax Tree example

3.2.1 Visitor Nodes

The JavaCC tools give the possibility of using Visitor classes, which, when implemented, they will visit the nodes of a parse tree sequentially, and interpret the tree structure. The Parser outputs a syntax tree built with visitor nodes, while JavaCC will have already generated a Visitor interface. Each visitor node will call a method when this is visited during the tree traversal, and the implementation of this visiting method is done within the visitor class that implements the Visitor interface. This means that more than one visitor class can be developed, and each of these can be specifically implemented to the requirements.

During our implementation a number of visitors were implemented to meet the specifications that were required. The following modules implement the visitor interface generated by JavaCC. This is given in Appendix C, Figure C.1.

Type-Checker traverses the syntax tree checking the channel types for arity mismatch. It implements the typing rules given in Table 2.6. Development details given in section 3.3.

Compiler will construct a machine representation based on the traversal of the syntax tree, but without enforcing checking on types. See section 3.4 for more detail.

GraphicTree Translator interprets and translates tree nodes into graphic nodes which can be drawn on a graphic context, thus, depicting the visual idea of a syntax tree. This graphic context can then be drawn onto a windows form to be viewed on screen.

3.3 The Type-Checker

The Type-Checker is a module which implements the Visitor interface, and its job is to type-check the channels. The Type-Checker implementation follows the typing rules given in Table 2.6. The Abstract Syntax Tree is given to the Type-Checker module, which will apply the syntax rules to the syntax tree. The Type-Checker module will then output a message indicating that the tree is correctly typed, or an error message, which will indicate where a typing mismatch occurred. The module does not alter the Abstract Syntax tree in any way. In fact, the Type-Checker can be skipped through the process, and allow the compiler to translate the syntax tree, regardless of its unstable state. This will allow the user to experience a runtime failure, if this is desired.

This module starts by interpreting the type declarations, and adding each declared type to an environment table. For every process that will be type checked, a copy of the current environment table is attached with the process. The environment table is implemented as a Hash Table where, the channel name, or the variable, will be the key element, while the referenced object will be the tree representation of the type (see page 19). The whole process will make use of a Stack, where the Hash Tables are stored before these are attached to type check a process.

Example 3.2 Controlling environments: Consider the following system, with a Hash Table for the type declarations Γ. These declarations are left unchanged before type checking each of the other processes by means of a stack.

$$\Gamma\ (P \mid Q \mid R)$$

The Type-Checker will first place the Hash Table on the stack, then it will take a copy and use it to type-check process P. During the checking of P, the Type-Checker will manipulate the entries as required. When the checking on process P ends, the table is discarded and a fresh copy of the original Hash Table is used for the type-checking of

process Q. The same procedure is used for all concurrent processes.

The stack will be really required when a process contains parenthesis, which will divide a single process into a number of other processes.

$$\Gamma\ (P.(M \mid N) \mid Q \mid R)$$

In this case the Hash Table that is used by process P is not discarded, but it is pushed on top of the stack, with all the changes that it had undertaken. This table will then be used by all the processes that have emerged from P (ie. M and N), using the same procedure. Upon the completion of P, the Hash Table will be popped from the stack and discarded, therefore the processes Q and R will have access to the original Hash Table that P had started off with.

3.4 The Compiler

The Compiler module translates an Abstract Syntax Tree into intermediate code. The compiler implements the Visitor interface constructed by the JavaCC tools, and it traverses the syntax tree using visiting nodes. During the traversal of the tree, the compiler will build data structures according to what the nodes of the syntax tree represent.

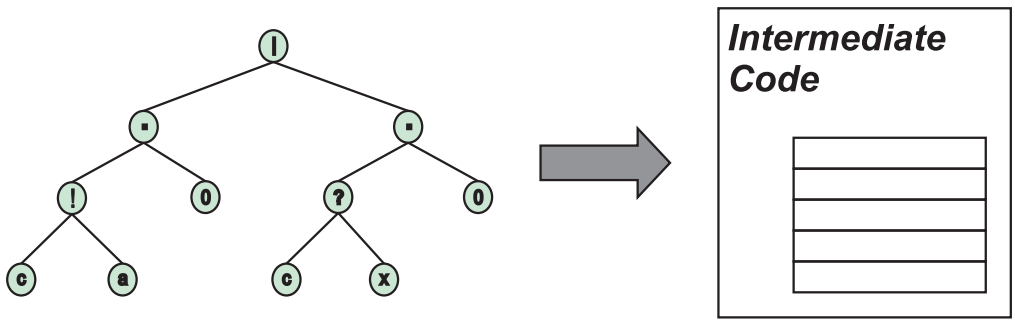


Figure 3.4: The compiler translates an AST into Intermediate Code

3.4.1 Intermediate Code Representation

At this stage, we shall define how the intermediate code representation is designed. This representation is developed such that, it can be saved (or serialized) to a file, making it the compiled version of the program.

A π-Calculus program is represented as an unordered list of processes, meaning that $P \mid Q \mid R$ will be represented as $[\!|\ P\ ::\ Q\ ::\ R\ |\!]$. An unordered list signifies that the processes of this list are randomly ordered, making sure that there will be as much fairness as possible during execution.

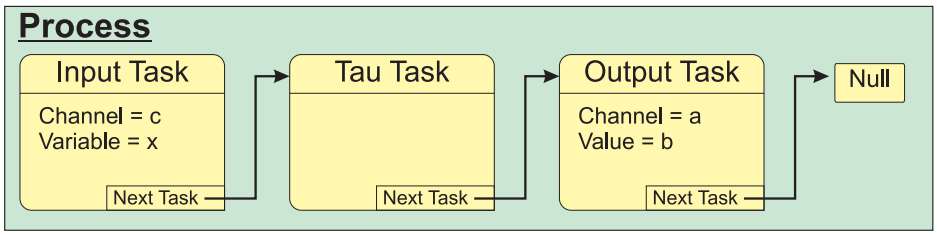


Figure 3.5: Intermediate representation of a process

On the other hand, the processes will be represented as an ordered list of π-*Calculus tasks*. In π-Calculus, a process is a sequence of actions. Hence in our machine we shall not define an entity to represent a process, but rather, we will build up the process by using a linked list of actions which we shall call tasks. We shall define a number of tasks, where each of these will correspond to the π-Calculus actions. Figure 3.6 gives the class diagram for the classes representing these tasks, each of which inherit from an abstract class Task.

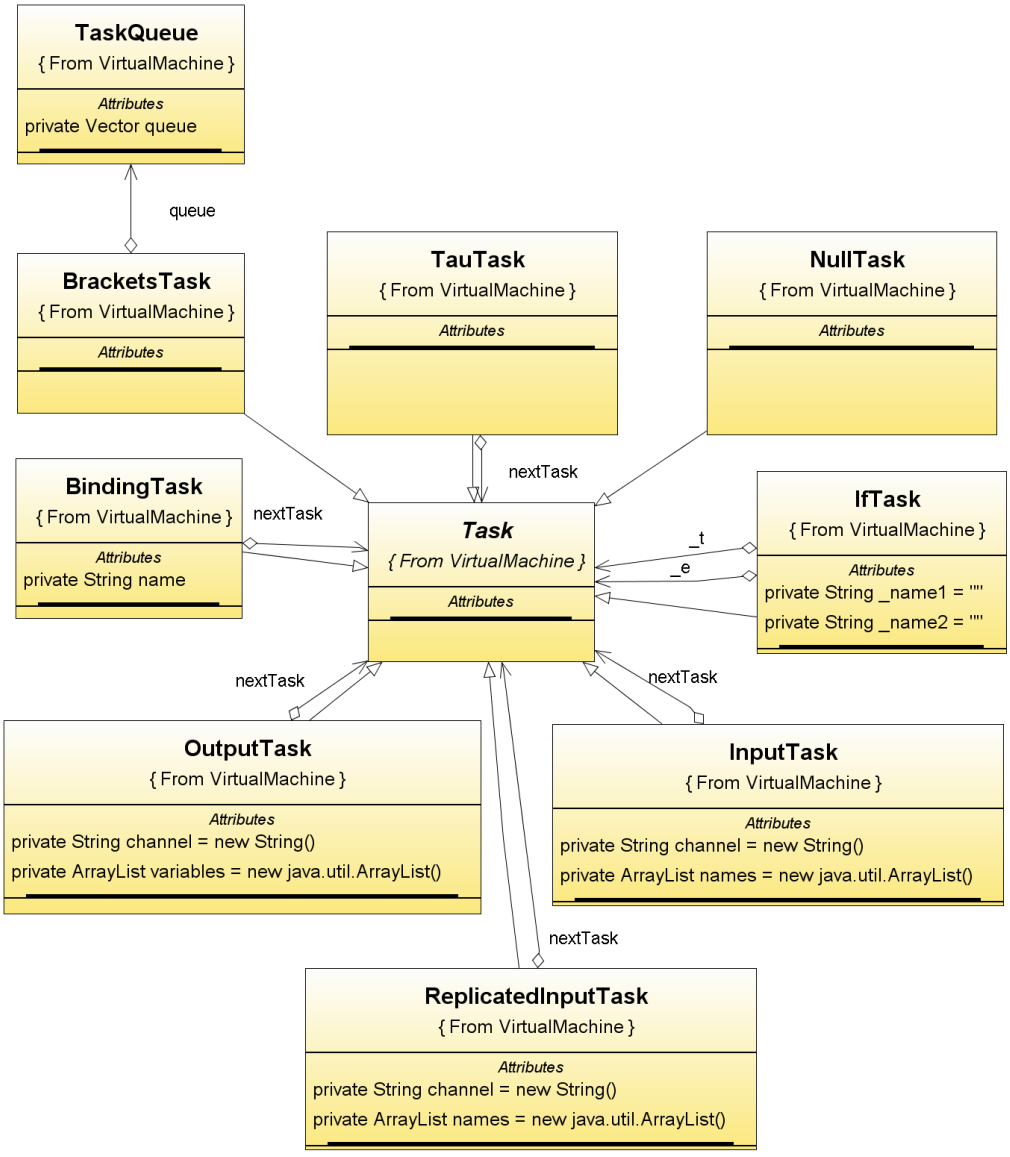


Figure 3.6: Class diagram - Task class and subclasses

Output Task

An Output task represents an output action, for example $c![x_1, ..., x_n].P$. This task will contain a string value for the channel c, and a list of strings for the channels $x_1, ..., x_n$ being outputted. This task must be part of a continuation or sequence of other actions, and therefore it must be followed by another task. The next task P will be represented as a task.

Input Task

An Input task represents an input action such as $c?(x_1, ..., x_n).P$. Hence, this task will contain a string value for the channel c, and a list of strings for the names $x_1, ..., x_n$. The following process P will be represented as a next task.

Replicated Input Task

A Replicated Input task is very similar to an input task, but it represents a replicated input action such as $*c?(x_1, ..., x_n).P$. This task will contain a string value for the channel c, and a list of strings for the names $x_1, ..., x_n$. The following process P will be represented as a next task.

Tau Task

The Tau Task corresponds to an internal action, or an unobserved action, $\&.P$. Hence this task will hold no information, other then the next task to be followed.

If Task

An If Task will stand for an If-Then-Else action. Let us consider the example $if\ x = y\ then\ P\ else\ Q$. This task will define two strings to hold the two channels,x and y, used for the equality. This task will store two pointers to two different tasks. One task is pointing to P, which is the task to be followed if the equality is true, and the other task is pointing to Q, the task to be followed if the equality is false.

Binding Task

A Binding Task is defined to correspond to the binding action of a channel. In order to accurately handle the binding and scoping of channels, a binding action or task had to be materialized to a specific point in time within the execution. This means that the syntax $(\#n)P$ will be represented by a single binding task. This task will contain a string to store the channel that is being bound, followed be the process with the bound channel, which is represented as a task.

Brackets Task

The Brackets Task is defined to be used for structural purposes only. During the first phases of the research it was noted that parentheses are used frequently to structure the π-Calculus syntax. For example, the system $P.(Q \mid R)$ is allowed, where the process P is followed by two unordered processes Q and R. Therefore, this will be represented as a *Brackets Task* which will contain an unordered list of unordered tasks.

Null Task

The Null Task is defined to represent the end of the link list of tasks. It is used for internal processing, but it can be compared to the null process 0.

3.4.2 Intermediate Code to String Translator

A translator module was developed to allow us to convert a compiled version of a π-Calculus program into a string representation. This module can be closely related to a de-compiler. This module, is a necessity for the Virtual Machines that will be developed since, the machines will have to output the resulting state, after performing a number of reduction steps. Therefore, this module will be called from the virtual machine, to translate the current machine representation into a readable format.

This translator will traverse a list of processes, and for each of the processes, the translator will traverse the tasks sequentially, and the corresponding string representation is composed.

3.5 A Minor Optimization

An optimization was adopted at this stage for the If-Then-Else operation. The idea is to type-check the channels involved in the condition expression. If these are not of the same type, then it can be deduced that the channels will never be equal. Therefore, the **Then** process will never be followed and thus, it can be removed. The following pseudo-code is adopted.

1. Retrieve type declaration for both channels, and check for the type equality.
2. If both channels are of the same type then construct the If Task normally.
3. If the two channels are not of the same type, then discard the Then section, and construct the Else process as a continuation to the preceding process.

Chapter 4

A Stand-Alone Virtual Machine

In Chapter 3 we have dealt with the design of the compiler, which is able to translate π-Calculus notation into machine data structures, such that this format is easier to manipulate and work with. The aim of this chapter is to develop a virtual machine based on the abstract machine given by Turner in his dissertation [Tur95]. The virtual machine performs reduction operations on the intermediate code, in the same way that π-Calculus notation is reduced using the reduction rules in Table 2.3. More specifically, we shall develop a set of reduction procedures, which will be applied to this intermediate code representation, allowing the processes to communicate.

This machine implementation is a Stand-Alone Virtual Machine, meaning that the whole execution will be done internally. This implies that the virtual machine will accept the input system as intermediate code, and will only output the results once that the computation has been terminated. The computation will be contained within the Stand-Alone Virtual Machine until the processes can no longer reduce among them, and it is then that the results are outputted. This brings us to an important objective, which involves the accuracy of the virtual machine. We want our stand-alone virtual machine to output correct

results when it terminates the execution. This means that the machine has to process the reduction rules consistently, and always produce the expected results. We shall tackle the correctness of our virtual machine in more detail in section 4.1.

A Stand-Alone Virtual machine gives us the advantage of focusing on the performance of the machine. Since all of the commutation will be under the control of the virtual machine, this allows us to develop an efficient architecture and expand on optimizations, which will perform rapid process reductions.

It is important to mention that the machine will be **simulating** the concurrent environment of the π-Calculus parallel processes, since it will be working on a uniprocessor (one processor). When mentioning a simulation, an essential point is brought forward; the **fairness** of the simulated execution. As we know, π-Calculus offers a non-deterministic execution, but, since our machine will run on a single processor, our machine will perform a deterministic execution. Hence, throughout the designing stages, it is extremely important to keep the handling and the execution of the processes as fair as possible. This involves having every process being given an equal chance for communication. This might turn out to be more difficult to accomplish than the actual correctness and accuracy of the machine, since it is easy to produce a correct result without achieving the correct amount of fairness between the processes. To summarize, our main objective will be develop a stand-alone virtual machine capable of executing the communication between processes and reducing these processes with a hundred percent correctness, while achieving this as fairly as possible.

4.1 Correctness Of The Virtual Machine

When talking about the correctness of a machine, we mean to say whether or not the machine matches reality or its realistic counterpart. To prove the correctness of our machine,

we shall compare the virtual machine results with the actual real results of π-Calculus, and make sure that the machine results are accurate. Since our Virtual Machine will be developed on algorithms, we can positively state that our machine is correct, if the algorithms that make up the machine are correct with respect to a number of specifications, where our specifications are the π-Calculus reduction rules given in Table 2.3.

In Chapter 3 we have seen how to translate π-Calculus terms into intermediate code representations, by means of the Compiler modules. We have to our availability, an *Intermediate Code to String Translator* (see section 3.4.2), which translates the machine representation into a string which is readable. Therefore, we affirm that, if we are able to translate π-Calculus terms into intermediate code and vice-versa, then the two forms can be regarded as equivalent.

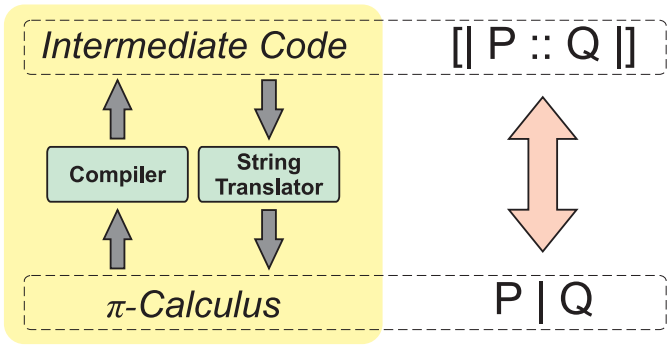


Figure 4.1: Converting π-Calculus terms to machine terms, and vice-versa

Consider the example $P \mid Q$ given in Figure 4.1. By using the Compiler, we translate our system into Intermediate Code Representation format, and by using the String Translator we convert the Intermediate Code into its original state.

Now,we can move a step further and define a function $fVM(S)$ within our Virtual Machine implementation, which will accept a π-Calculus system S in intermediate code format, and outputs the resulting system in intermediate code. Internally this function will consist of all the rules discussed in Table 2.3.

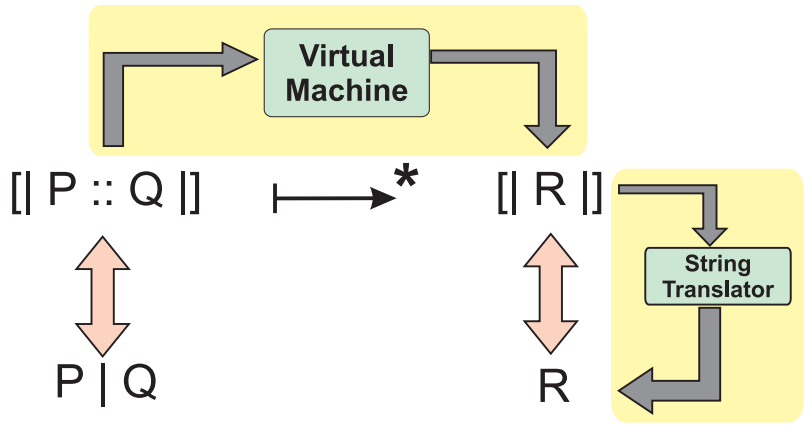


Figure 4.2: Correctness of machine

Next, we utilize our function (Virtual Machine) as shown in Figure 4.2, which will reduce the intermediate code representation in a number of steps as follows:

$$fVM([|\ P :: Q\ |]) \longrightarrow^* [|\ R\ |]$$

Now we need to translate the results back into π-Calculus terms, and for this, we use our Intermediate Code to String translator, as depicted in Figure 4.2.

So far we have achieved a result using the Virtual Machine. Consequently, we now have to determine the correctness of the Virtual Machine. In order to determine this, we have to show that the acquired result is equivalent to its realistic counterpart result. By the realistic result we refer to the result achieved if the starting system had to be reduced by using the reduction rules in Table 2.3.

This is clearly pictured in Figure 4.3. Please note that the results of the reduction transitions of the π-Calculus terms and the results of reduction transitions of the Intermediate Code, can be achieved differently.

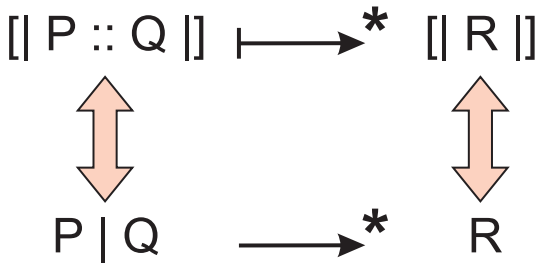


Figure 4.3: Correctness of machine

4.2 Stand-Alone Architecture

The design for the Stand-Alone Virtual Machine is based on the machine described by David Turner in his thesis [Tur95].

The machine consists of a *Process Manager*, which will contain instructions for process reduction or *Reduction Procedures*. This module will work in between two First In First Out (FIFO) queues, the *Run Queue* and the *Service Heap*. As seen in Figure 4.4, the processes[1] will circulate around the two queues and are handled by the *Process Manager*.

When the virtual machine commences its execution, it will have available an unorder list of processes, as described in Section 3.4.1. These processes will be randomly enqueued onto the Run Queue, thus giving the simulation a higher level of fairness. Once that all the processes are enqueued, the simulation can commence.

At this stage, it is important to make a distinction between the nature of the tasks involved in the processes. We classify Output Tasks, Input Tasks and Replicated Input Tasks as **communication tasks**, since these tasks represent the actions that are always handled in pairs, because communication must be between two processes. The rest of the tasks; the Tau Tasks, the Binding Tasks, the Brackets Tasks, the If Tasks and the Null Tasks are **non-communication tasks**, meaning that these represent procedures that do not require

[1] A single process is actually a linked list of tasks. See Section 3.4.1.

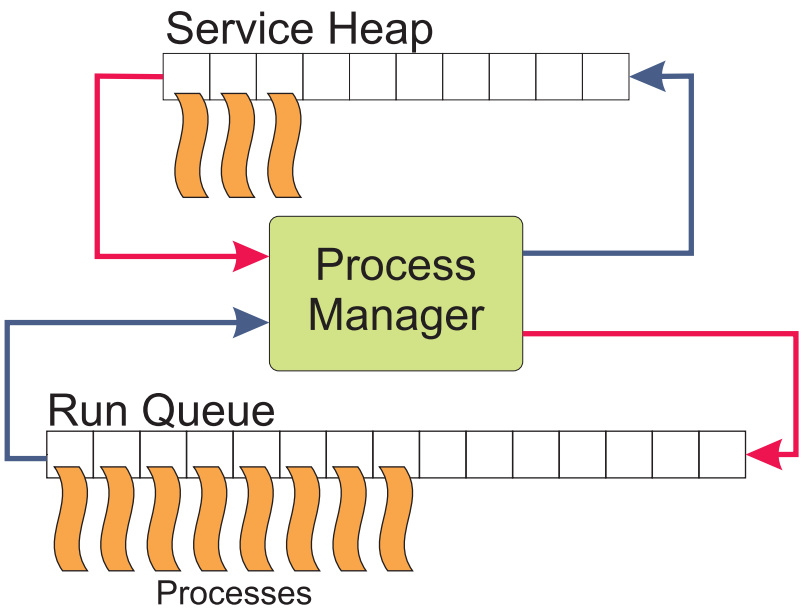


Figure 4.4: Simple design for the Stand-Alone Virtual Machine

another process to be executed.

The process manager will, retrieve the process at the head of the queue, and it will handle the first task of the process, according to the classification of this task. The difference in handling, is that, communication tasks make use of the Service Heap, while non-communication tasks do not. The account for communication tasks is given in Section 4.3, and the account for non-communication tasks is given in Section 4.4.

4.3 Handling Process Communication

If the *Process Manager* dequeues a communication task, such as Input, Output or a Replicated Input from the Run Queue, the Process Manager will follow the flowchart depicted in Figure 4.5.

The Process Manager will have a single process at a time to handle. Thus, only one task

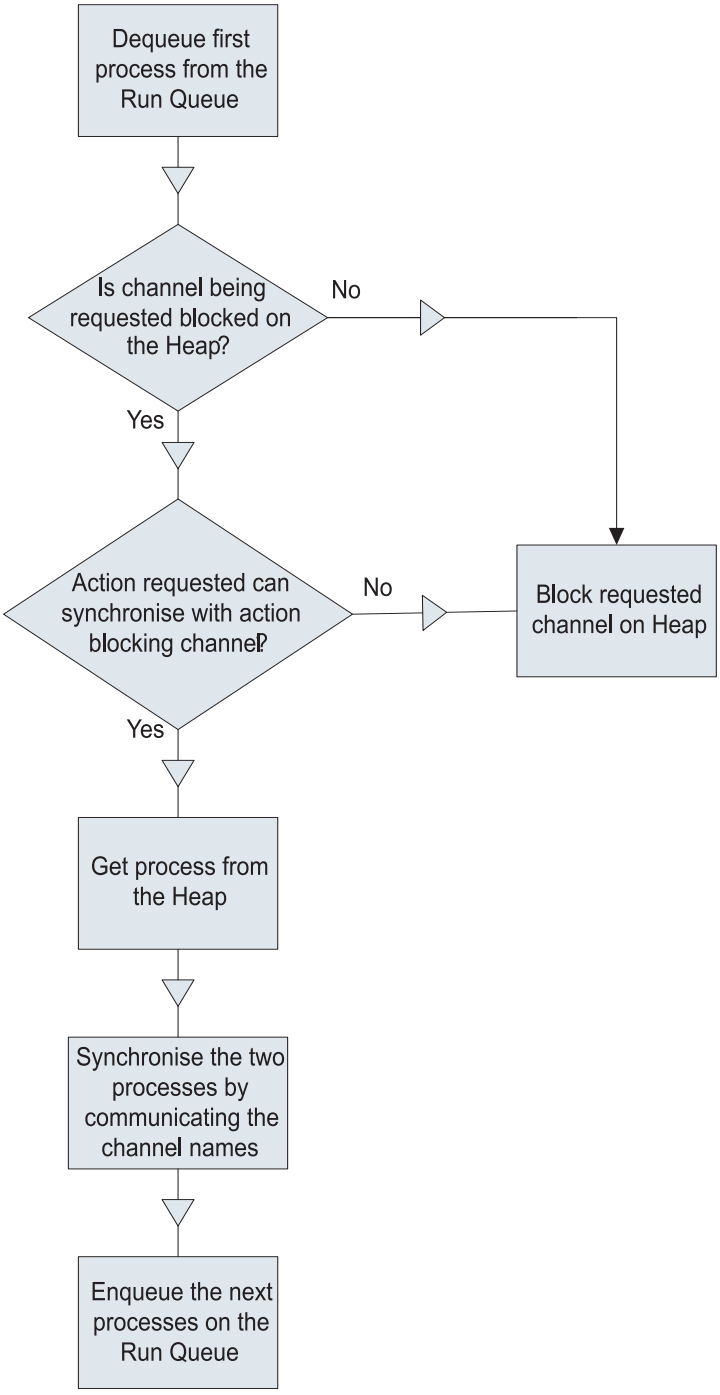


Figure 4.5: Stand-Alone Machine algorithm

needs to be handled. Communication tasks will always request a channel to communicate through,

Example 4.1: For instance, all of the following processes are requesting the channel c.

$$c![a, b, d].P | * c![b, d, e].Q | c?(x, y, z).R$$

The Process Manager will search for a task along the Service Heap, which had requested the same channel. This is known as channel blocking, since the search to match the pair is only done on the heap, and thus having the effect of the channels being blocked by the requiting task.

If a task is not found in the heap, the requested channel is not blocked, and no other process has requested this channel. Hence, the Process Manager will enqueue the process on the Service Heap, and thus blocking the channel which is being requested.

On the other hand, if a process is found blocking a channel on the Service Heap, the Process Manager will check the **polarity** of the two tasks. This means that the Process Manager will compare the task being serviced with the one found on the heap, and will match up a pair only if the two tasks can communicate as specified in Section 2.1.2. For instance, an output is matched with either an input or a replicated input. If the other half of the communication pair is not available, then the task being serviced is enqueued on the Service Heap.

Nonetheless, if a match is found, then the two processes can communicate and reduce. Hence the Process Manager will retrieve the names from the output action and substitute the variables of the input action, with the retrieved channel names. If type-checking has been performed during compilation, then the number of names being passed will match up with the number of variables. However, if the Type-Checker module has been over-passed, then the Process Manager will halt and trigger a runtime failure.

Following the communication, the two processes are reduced, by discarding the first task of each process, since these tasks have been performed. Afterwards, these processes are enqueued to the end of the Run Queue, and subsequently, the Process Manager commences this procedure once again.

In this whole procedure, the processes are being circulated around as much as possible so that we try to account for fairness as mush as possible. The Service Heap acts as a meeting point for the processes, avoiding the situation of having two processes which never pair up to communicate.

Example 4.2: The following example shows how two processes are enqueued on the Run Queue, and how the Process Manager handles these processes, by using the Service Heap to establish communication.

Process Manager	*Run Queue*	*Service Heap*
$_P\langle \diamond \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle\ c![a,b].Q\ \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle\ c![a,b].Q \mid c?(x,y).R\ \rangle$	$_H\langle \diamond \rangle$
$_P\langle\ c![a,b].Q\ \rangle$	$_R\langle\ c?(x,y).R\ \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle\ c?(x,y).R\ \rangle$	$_H\langle\ c![a,b].Q\ \rangle$
$_P\langle\ c?(x,y).R\ \rangle$	$_R\langle \diamond \rangle$	$_H\langle\ c![a,b].Q\ \rangle$
$_P\langle\ c?(x,y).R \bullet c![a,b].Q\ \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle\ R\{^{a,b}/_{x,y}\} \bullet Q\ \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle\ Q\ \rangle$	$_R\langle\ R\{^{a,b}/_{x,y}\}\ \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle\ R\{^{a,b}/_{x,y}\} \mid Q\ \rangle$	$_H\langle \diamond \rangle$

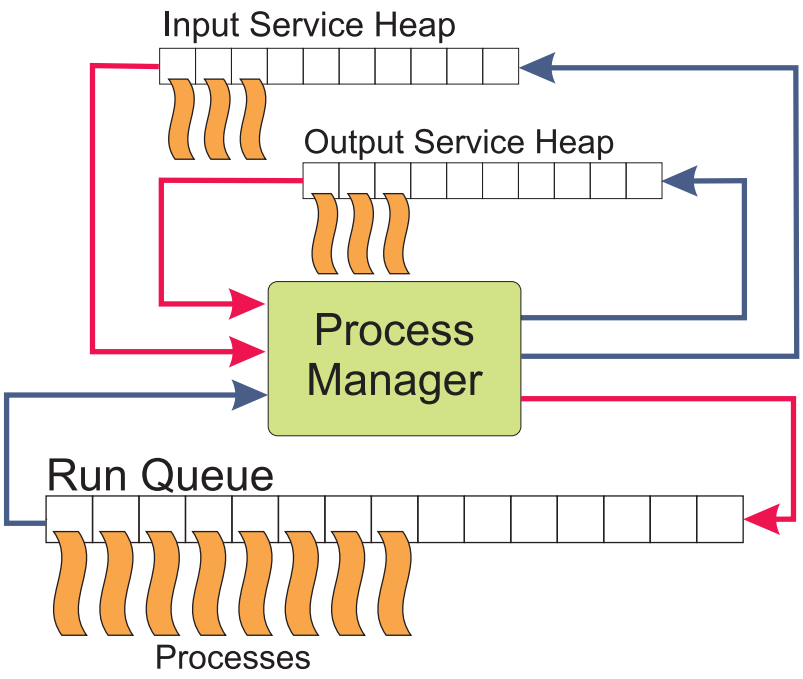


Figure 4.6: Stand-Alone machine with optimized service heap

4.3.1 Optimizing The Service Heap

An important optimization that is implemented to this mechanism, is the division of the Service Heap into two sections, one for output tasks and another for input tasks. This optimization is a variation to the optimization implemented by Turner. The idea is to have two Service Heaps, rather then a single one, where input tasks and replicated input tasks are separated from output tasks, reducing the search space by half[2]. So now, the algorithm will work similar to the one described in Section 4.3, with the only difference that, if an output is being serviced, then, the search is done only on the Input Service Heap, and vice-versa for inputs. If the process needs to be enqueued to the Service Heap, then this is enqueued to the Input Service Heap, whether its task is an Input or a Replicated Input. If the task is an Output then this is enqueued to the Output Service Heap.

[2]This is the worst case scenario where $\frac{1}{2}$ of the tasks are outputs and the other $\frac{1}{2}$ are a mixture of inputs and replicated inputs

Replicated Input tasks are treated in the same manner as Input tasks, throughout the handling of processes. Some researchers suggest that Replicated Inputs should be allocated on a separate heap, which will be a permanent storage for these processes. This would be a good optimization, but it was not implemented here. The reason for this is that if a Replicated Input had to be stored on a permanent Service Heap, then this will permanently block a channel. Thus, it will have priority over other processes (Input Tasks), having a fairness breach during the execution of the simulation. Hence, it was decided, that, in our implementation, the machine will have to absorb the cost of enqueues/dequeues of Replicated Input tasks, but gaining the fairness of the execution.

4.4 Handling Non-Communication Tasks

The rest of the tasks, such as Tau Task, If Task, Binding Task, Brackets Task and Null Task, are processed differently, since these can reduce a process independently and without using the Service Heap.

The **Tau Task** serves only as an internal operation that is performed within the process. Hence, the Processor does not need to perform any additional handling of this task. The procedure is that the next task is simply enqueued at the end of the Run Queue.

Example 4.3 (Handling Tau Tasks):

$Process\ Manager$	$Run\ Queue$	$Service\ Heap$
$_P\langle \diamond \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle \tau.P \rangle$	$_H\langle \diamond \rangle$
$_P\langle \tau.P \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle P \rangle$	$_R\langle \diamond \rangle$	$_H\langle \diamond \rangle$
$_P\langle \diamond \rangle$	$_R\langle P \rangle$	$_H\langle \diamond \rangle$

An **If Task** is handled by the Processor by evaluating the conditional equality that it contains. If the equality is evaluated to a true value, then the process for the *then* part is enqueued, while if the equality is calculated to a false value, then the process for the *else* part is enqueued.

Example 4.4 (Handling If Tasks):

$Process\ Manager$	$Run\ Queue$	$Service\ Heap$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ [x=y](P)(Q)\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \underbrace{[x=y]}_{true}(P)(Q)\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ P\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ P\ \rangle$	${}_H\langle\ \diamond\ \rangle$

The **Binding Task** purpose is to bind the given name to the process. In order to accomplish this, the Processor will issue a fresh name, which is unique throughout the whole system, and it will substitute the channel name that was to be bound, with this new fresh name. This ensures that the name is unique to the process, which has the same effect as being bound. The Processor will then enqueue the process at the end of the Run Queue.

Example 4.5 (Handling Binding Tasks):

$Process\ Manager$	$Run\ Queue$	$Service\ Heap$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ (\#n)P\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ (\#n)P\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ P\{{}^{n0}/_{n}\}\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ P\{{}^{n0}/_{n}\}\ \rangle$	${}_H\langle\ \diamond\ \rangle$

Brackets Tasks contain a list of processes, hence the Processor's job is to enqueue each of these processes to the Run Queue. The idea is to open up the brackets and release the processes that were enclosed as new processes.

Example 4.6 (Handling Brackets Tasks):

$Process\ Manager$	$Run\ Queue$	$Service\ Heap$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ (M \mid N)\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ (M \mid N)\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ M \bullet N\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ N\ \rangle$	${}_R\langle\ M\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ M \mid N\ \rangle$	${}_H\langle\ \diamond\ \rangle$

The **Null Task** indicates to the Processor the end of a process, or a Null process. Hence, the Processor will simply discard the instance, and returns to servicing other processes.

Example 4.7 (Handling Null Tasks):

$Process\ Manager$	$Run\ Queue$	$Service\ Heap$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ 0\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ 0\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$
${}_P\langle\ \diamond\ \rangle$	${}_R\langle\ \diamond\ \rangle$	${}_H\langle\ \diamond\ \rangle$

4.4.1 Optimizing The Management Of Tasks

We employ a small change while handling the processes. The optimization concerns the non-communication tasks which are discussed in Section 4.4. As one can notice, these tasks can avoid being queued up on the Run Queue.

Example 4.8: Consider the following process as the simplest of examples. It would be good practice if our processor will execute all of the internal actions and enqueue the process P at once, instead of queuing the process after each continuation.

$$\&.\&.\&.P$$

In our implementation of the machine we shall allow only the queuing of Input tasks, Output tasks and Replicated Input tasks, (Communication Tasks), while all of the other tasks have to be processed by a filtering module, before being queued. The *Filter* will perform this processing until a communication task is available, after which this will be enqueued to the Run Queue.

By introducing this policy, we are making the execution more fair between processes.

Example 4.9: For instance, note the following state on the Run Queue.

$$\langle\, c![a].P \mid (\#z)c?(x).z![x].Q \mid c?(y).R \,\rangle$$

If we do not apply the optimization that is being suggested, then the third process has an advantage over the second process, because the second process is still waiting to process the binding action, which does not affect, in any way, the following action. Therefore, by processing a non-communication task, we make sure that all processes have a more just chance of execution.

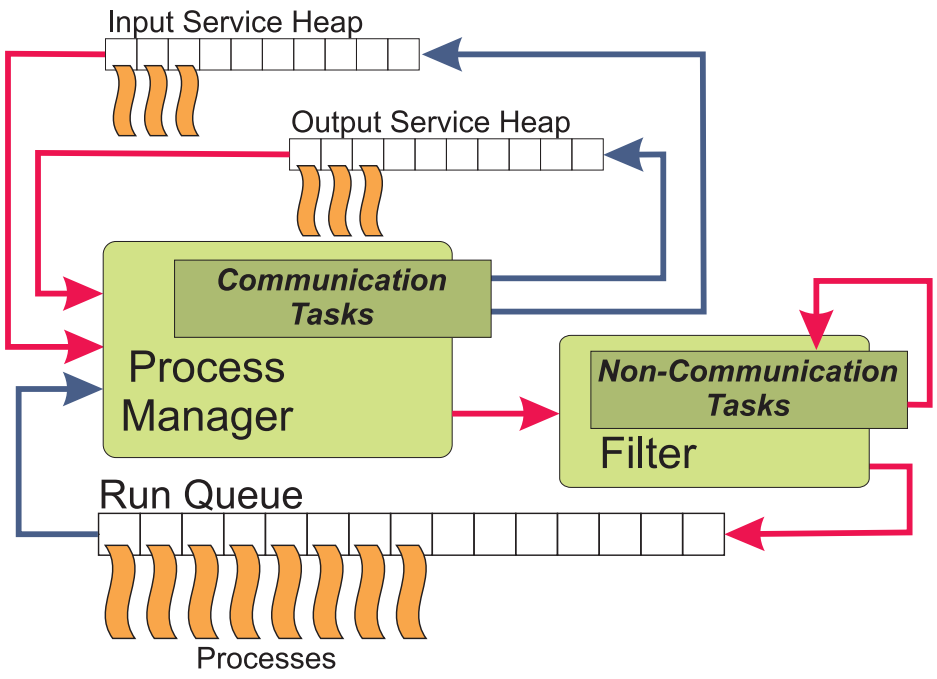


Figure 4.7: Stand-Alone machine applying all the discussed optimizations

4.5 Environments

Performing substitutions within processes during communication, is a very expensive procedure. Apart from being costly, we are never sure if a process will survive through its whole lifetime. Therefore, if a process had to be stopped for some reason, then all of the substitutions that were performed, will be useless and waisted. We improve this situation by introducing *Environments*. Environments permit us to avoid substitutions completely, meaning that the variable terms are never modified within the processes.

An environment E is the mapping from a variable x to a channel name c.

$$E ::= x \mapsto c$$

These mappings are stored using a Hash Table. Note that environments are bound to the usage of a single process Hence we modify our virtual machine, such that we can store

an environment table (Hash Table) for each of the processes that the machine will be handling.

Therefore, when a substitution is required, instead of performing a direct substitution $\{^{new}/_{old}\}$, a new entry is added to the process' environment table $old \mapsto new$. Thus, whenever it is required to use the *old* variable as a channel, this is looked up in the hash table and the correct channel name is retrieved.

Chapter 5

An Interactive Virtual Machine

In this chapter we discuss the development of a virtual machine which will allow a user to interact with the machine itself, while this performs the same functionalities as those within the Stand-Alone Virtual Machine. By this we understand that the Interactive Virtual Machine will first have to reduce a given π-Calculus program, after which the results are shown to the user. Based on these results the user will input π-Calculus commands, which these will trigger the virtual machine to reduce the program even further, thus continuing with the execution. After that the program is again reduced, the machine will display to the user the next results, with which the user can again interact.

The idea is to have a user inputting a π-Calculus program, and while the virtual machine is executing this program, the user can communicate with the program using π-Calculus notation. The user will be allowed to compose Input Tasks and Output Tasks, and use these to communicate with the internal program. This means that, in π-Calculus terms the user, together with these actions that are being used, can be regarded as a process. The user is an external process, working concurrently with the virtual machine, and communicating with it. We see this in Figure 5.1 where process U is the user and process M is the virtual

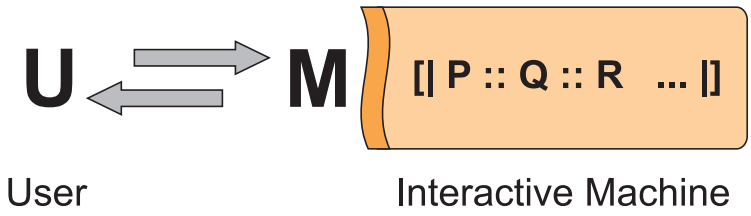


Figure 5.1: User interacting with machine

machine. Note how the machine is not a single process, but rather it is composed of the internal process which make up the π-Calculus program.

This machine focuses mostly with the communication between the user and the machine, however we will allow communication between the internal processes of the program which is being executed. In other words we have to provide two modes of operation for this virtual machine.

Stepped Into - when the machine stops for user interaction at every single reduction. The machine outputs its current state to the user, at every step of the execution, even if this is performed internally between the program processes.

Stepped Over - the machine will only halt when no more internal reductions are available. This allows the internal program to process logical work that is to be performed, before interacting with the user.

5.1 Stand-Alone vs Interactive

Let us investigate on what an Interactive Machine will serve for, and why it is an important tool to be developed and analyzed. Note how an Interactive environment, offers many more interesting uses than the Stand-Alone environment. But what are the advantages of developing an Interactive Virtual Machine over a Stand-Alone Virtual Machine?

Note that the Stand-Alone Virtual Machine developed in Chapter 4 offers limited functionalities. Programs running within the Stand-Alone machine are restricted only to the environment offered by the machine. This makes it hard for the user to assess the program's behavior. In fact, the Stand-Alone Virtual Machine does not give the user the desired perspective and thus the user cannot analyze the program internal reductions.

The most straightforward use of having an Interactive Virtual Machine is to give the user the possibility to understand and learn the π-Calculus notation. The machine can serve as a learning tool for those who wish to learn the π-Calculus, since the reductions can be visualized more then when giving examples on paper. The idea is to aid the learning process since the user can feel part of the π-Calculus system.

Another useful job that an Interactive Virtual Machine can accomplish, is to debug π-Calculus programs. The interaction feature can also act as a tracing functionality to the program that is being executed. During interaction, the developer can learn of mistakes, or bugs, within the program. This will be possible since the user will have a clear picture of the current state of the machine.

A more interesting observation is that by developing such a virtual machine, we are creating a level of abstraction on top of the π-Calculus program. Similar to what Sewell accomplished in the development of the Nomadic-PICT [PSP99, SW]. Indirectly, the framework now has a layer of abstraction where the execution of the internal π-Calculus program is abstracted away from the user, by means of the interactive virtual machine. The user will still be communicating with the program using π-Calculus notation, however the communication is done through the virtual machine, thus the details of the internal program is unobservable from the user. The interesting point is that, both layers are using the same π-Calculus notation. As we shall see in Chapter 6, when we test the implementation of this machine, is that, if we position a user in front of two π-Calculus programs which implement the same interface (input/output relations) and functionality, then the user will

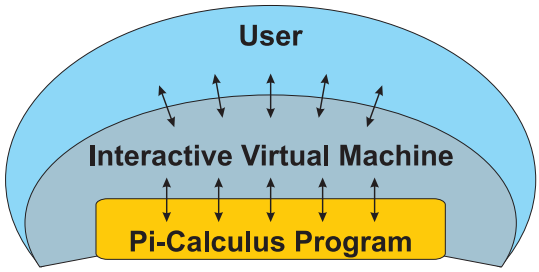


Figure 5.2: Abstraction layer provided by the Interactive Virtual Machine

not be able to distinguish one from the other despite that these are internally implemented differently.

At this point we have started to deviate beyond the scope of this study. However, it is extremely motivating, how such an interactive machine when correctly developed, will transform the environment to a more realistic distributed system. This is so, since an interactive machine will not only work on a single processor but it is now communicating with an external source, in this case the user. An interesting question will be, what if the interaction had to be performed between two or more interactive machines?

In this kind of machine we see that our priorities change, since at this point, the machine does not only have to perform efficiently with its inner processing, but also in retrieving information about the current state of the machine. The machine state has to be retrieved and showed to the user as efficient as possible. In this machine the main processing power has to tackle the crossing point between the User and the Machine. We need to assume that the user will be a slow process, so we have to bargain with the internal structure of the machine, to make the User-Machine interface more efficient. By applying this change in performance priority we need to apply changes to the internal design of the machine.

Fairness is another vital point in this machine implementation, since now, we have the user acting as an external process, which will have priority at certain times. For this we

have to examine the two modes of interaction. If we take the Stepped Into[1] mode then the fairness of the overall machine will be reduced, since the user will have priority over all the other processes. However, note that when the user chooses an internal action to be processed, then this choice has to be randomly selected from the set of internal actions. Therefore, it is interesting to note that in this respect we have to take into account the fairness of the execution.

On the other hand, when the machine will be operating in Stepped Over[2] mode, the fairness of the machine will have to be accounted for in a similar way as within the stand-alone machine. This means that the user will actually not be given any preference when operating in this mode. This is an important point to emphasis on, since the machine will allow interaction when no additional reductions are possible to be processed. Hence, as we have already discussed in Chapter 4, all the internal processing of the machine will execute on a rational framework.

5.2 Correctness Of The Virtual Machine

For this machine we still need to take into account the correctness of the virtual machine. The objective is very similar to the correctness of the stand-alone machine (see Section 4.1). In fact the underlying intentions do not change, meaning that, when a system is given to the machine, the result that is produced has to be equivalent to its realistic counterpart. The only difference here, is that, the correctness has to be monitored iteratively, each time the user interacts with the machine. It is very important to have the correctness of the interactive machine stable throughout the whole execution since, the results produced by the machine will reflect the input that is to follow. Therefore, the interactive machine we will keep up with our previous objective regarding correctness.

[1]The machine stops and waits for user interaction at every step, even at internal tasks.

[2]The machine will stop and wait for user interaction when no further internal reductions will be possible.

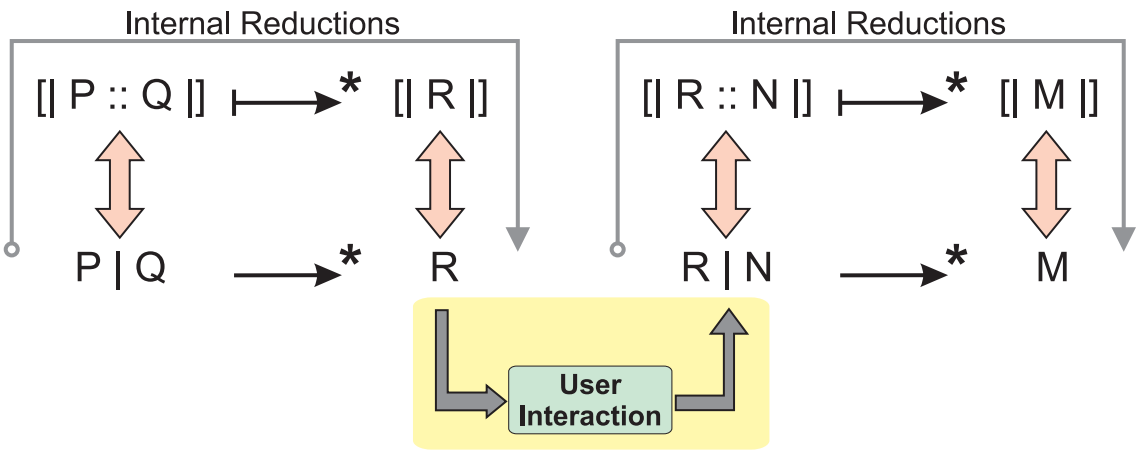


Figure 5.3: Correctness of the Interactive Machine

Figure 5.3 shows us the correctness of the virtual machine. Note how the execution of this virtual machine, is engaged into an iterative process and the internal processing is trigged upon user interaction. Therefore, the virtual machine will have to produce accurate results at each iteration. It is important to understand that the results, depend on the previous user input, hence the machine has to produce correct results with respect to the inputs that the user has done.

As we shall see, in order to have control over what the user inputs into the machine, we will use channel typing. Channel types will allow the machine to type-check the actions that are given to the machine by the user. If the action is not correctly typed, then the machine will complain to the user, thus avoiding arity mismatch at runtime.

5.3 Interactive Architecture

The development of an Interactive Virtual Machine, will mostly involve, an efficient mechanism which allows us to read the current machine status. This machine will be displaying the current state of the internal program, at every step during the simulated

execution. Hence, it is crucial to have optimal performance, in the linkage between the user and the machine.

We start by re-organizing the architecture of how processes are handled. Note that, the structure of the processes is the same as in the Stand-Alone Virtual Machine. This means that we will be operating on the same intermediate code representation, as discussed in Section 3.4.1.

The architecture design for the Interactive Virtual Machine is given in Figure 5.4. We have a *Process Manager*, which will be handling all process communication. By observing this design, one can note that, some of the strategies are similar to the Stand-Alone Virtual Machine.

We make a distinction between **Communication Tasks** and **Non-Communication Tasks**, after which we employ a further step to make another distinction between the channel requests of the communication tasks. This means that now, we are categorizing the communication tasks, by the channel name that is being requested.

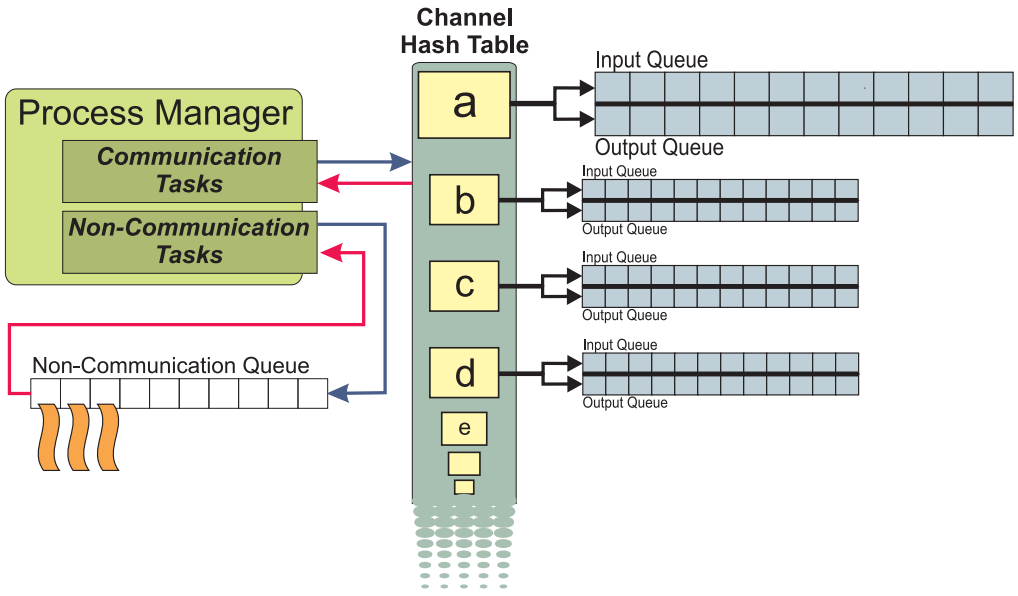


Figure 5.4: Interactive Virtual Machine Architecture

To accomplish this, we define an object (*Channel*), which represents the actual channel of communication, consisting mainly of an input queue and an output queue. The Process Manager which will enqueue processes on the respective queue. Several of these channels will be stored using a Hash Table, which allows easy and efficient access. Thus, we now have a design which will allow us to retrieve the current state of the machine very effectively and with the least amount of processing costs. For example, if we had to list the key elements of the Hash Table we immediately have all the channel names that are being used by the machine.

5.3.1 Channels

A Channel is composed of a name, identifying the channel, and a channel type object classifying the channel to a particular type definition. A channel will contain two process queues; one will queue input task and replicated input tasks, while the other will queue output tasks. These queues are dedicated queues, meaning that they will only enqueue processes that are requesting the same channel.

Note that the queues with this channel object, are similar to the Service Heap of the Stand-Alone Virtual Machine. In fact, we employed the first optimization, by distinguishing the input queue from the output queue, from the very start.

Criteria are defined for processes requesting to queue up for the use of a channel object. The first is that only processes with a communication task[3] can be enqueued. Another criteria is that, the channel being requested and the channel used by the task, have to be the same one, meaning that both the name and the channel type have to be equivalent. This ensures that runtime errors are not introduced by the user during interaction.

[3] A communication task can be either an Input Task, a Replicated Input Task or an Output Task.

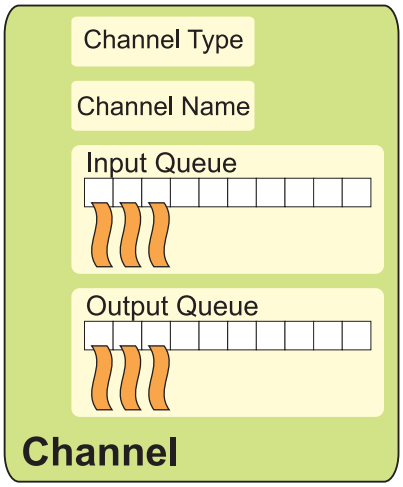


Figure 5.5: Channel Architecture

The most important function that a channel object can perform, is that a channel object can output its current state. The state of the channel is what is displayed to the user, and it should give an overview of how the user can interact with this particular channel. Therefore, the state of a channel is simply two boolean values, representing the two queues for inputs and outputs. These values will indicate if a channel has available processes awaiting in the input and output queues, hence, a true value means that at least one process is waiting in that queue, while a false value means that the queue is empty. This allows the machine to communicate to the user, the different combinations of a channel state.

5.4 Handling Process Communication

The communication between the processes is handled by the Process Manager. At this stage it is important to mention that in this virtual machine we will be dealing with two situation of process communication. Either a machine process communicates with the user during interaction, or two machine processes communicate internally. For the user to communicate with the machine, the user will have to create a task, which this will be

handled as the process which is to be serviced, meaning that communication will have to be done with this process if possible. On the other hand, if two of the machine processes will communicate internally between them, then from the user's point of view this will be regarded as an internal reduction.

When a user launches a task into the machine, the Process Manager selects the channel that is being requested. The channel retrieved is first checked for type equivalence. Once that the channel is identified as correct, the Process Manager will retrieve the first process awaiting to communicate from the channel. Now, if the user's task is an input task or a replicated input task, then the first process on the output queue is retrieved. On the other hand, if the user's task is an output task, then the first process from the input queue is retrieved. The Process Manager will then reduce the two processes in the same manner as in the Stand-Alone Virtual Machine, thus following the rules in Table 2.3.

For internal communication, the Process Manager will randomly select a channel which has non-empty queues (input and output). By having a channel with at least one task enqueued on each of the queues, then this signifies that the channel can communicate internally. Hence, the process manager will retrieve these processes and communication is accomplished.

5.4.1 Internal Reduction

When discussing internal reductions within the Interactive machine, we mean that the machine will either reduce a non-communication task, or that the machine will allow two processes having a matching pair of communication tasks, to communicate internally. Therefore, it is important to note that, from the user's point of view both of these situations are regarded as internal reductions.

Recall that we are dealing with two modes of interactivity within this virtual machine. When executing a Stepped Into simulation, then we need to allow the user to experience even internal reduction happening within the machine's program. Hence, the optimization employed for the Stand-Alone machine, is not applicable in this case. However, if a Stepped Over simulation is being run, then it is important to ensure that all internal reductions are performed.

To handle the Stepped Into simulation, we enforce that only one reduction is performed at every user interaction. This means that the Non-Communication Queue seen in Figure 5.4 will enqueue processes awaiting for a non-communication task to be performed. A channel which can perform an internal communication will be suppressed from reducing. Therefore, we then give the user the option to either interact with the machine by introducing a task, or by instructing the machine to perform internal tasks. When performing internal reductions within the machine, the user will still be able to notice the changes occurring in the program.

During Stepped Over execution, the interactive machine will check for both types of internal reductions and keep reducing until no further are available. This means that for every single user interaction, the virtual machine can possibly perform a number of reductions. Please note, that during this mode of operation, the virtual machine will not be utilizing the Non-Communication Queue since, these will not be allow to stand idle within the system.

5.5 Environments

In the implementation of the Interactive Virtual Machine, we make use of channel name environments just as these are used in the Stand-Alone machine. Therefore, we extend our channel object representation, and allow it to hold a set of name mappings for each of

the processes that are enqueued.

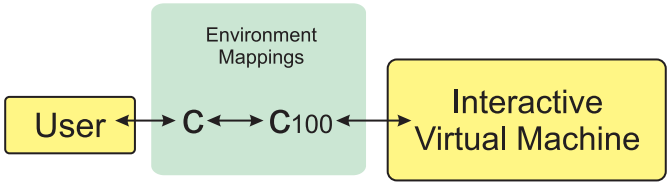


Figure 5.6: Environment Mappings between the user and the virtual machine

The difference that is required for the Interactive machine is that we have to keep a set of name environments for the user. Remember that the user is regarded as an external process, hence the machine will have an environment table, which will map the channel names which the user is using to the real channel names within the machine. As Figure 5.6 shows, the environments' mappings, are used to link the user's channel names to the appropriate channel name within the virtual machine.

5.6 Graphical User Interface Design

Throughout this project the GUI (Graphical User Interface) of the application was not part of the main objectives. However, since we have discussed the development of an Interactive Virtual Machine, we now realize that for an interactive machine to be easily understandable by the user, we require a user-friendly presentation.

Most important of all is how the GUI will present the virtual machine status. The most suitable solution was to design a table as depicted in Figure 5.7. This will list all of the free channels that the machine currently has available. The list will also give the type of the channel, thus the user will know how to handle communication with every listed channel. The channel rows have two columns representing the two queues for the channel. Checkboxes can be used to indicate whether processes are awaiting or not, within the channel queues.

Channel Name	Channel Type	Awaiting to Input	Awaiting to Output
a	<< >>	☑	☑
b	< <>, <> >	☐	☑
c	<>	☑	☐
d	< << >>, <> >	☐	☐
e	< >	☑	☐
f	< >	☐	☑
g	< V >	☐	☑
h	< < >, < > >	☑	☐
i	< >	☑	☑

Figure 5.7: Proposed table showing the current state of the machine

A panel will allow the user to create a task, and process this task within the virtual machine. Thus, on doing so, the status table will change the values showing how the system state has changed with the effect of the given task.

All of the other modules that we discussed will be incorporated within a simple application, which will serve as an Integrated Development Environment for π-Calculus. The IDE will offer an editor where π-Calculus programs can be written using the Π-Language, and options will be available to simulate the program on the two virtual machines that we discussed.

Chapter 6

Evaluation

This chapter mostly consist of testing analysis on the virtual machines that have been developed. As one can note, we emphasized on the correctness of our virtual machines. Now we get to evaluate the correctness of both machines. In Chapters 4 and 5, we mention correctness before the development, so that throughout the implementation our mind was focused on the construction of correct virtual machines, which produces accurate results. In this chapter we tackle the final part of machine correctness. Therefore, we have to verify, that both of our machines are indeed simulating how processes communicate, as defined in Table 2.3.

Figure 6.1 shows clearly our task in this chapter. Note that we will be focusing on the equivalence of the results. This means that we will compose π-Calculus examples, and simulate the execution of these examples on our virtual machines, thus obtain the first result. Hence, we will follow through the same examples on paper, to obtain a second result, we check for the equality of these results and if these are equivalent then this shows that the virtual machine is correct. To prove the correctness of the virtual machines we would have to test each of the different situations that the machines can end up in.

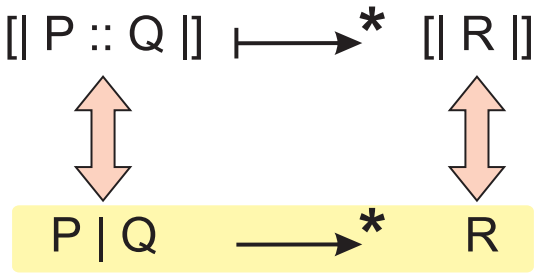


Figure 6.1: Evaluating the correctness of the virtual machines

The plan for the testing stages, is to, first test our Stand-Alone Virtual Machine implementation, followed by, testing the Interactive Virtual Machine using the same examples. This will allow us to verify that both machines are correct as well as consistent with each other. We then move on to, test the abstract layer of the Interactive Virtual Machine by composing examples, with different implementations, and investigate whether the user is able to distinguish between the examples.

6.1 Testing the Stand-Alone Virtual Machine

The Stand-Alone virtual machine was first tested for the basics of the π-Calculus actions, after which more complex examples where used for the testing. Note that the simple tests where omitted from this chapter, but we will be dealing with a number of elaborated examples. These examples offer all of the required test cases for our Stand-Alone virtual machine.

6.1.1 Test case - Memory cell

The first example is the same one given in Example 2.13 of Chapter 2. This example consists of a process capable of generating memory cells, and an other process that is

utilizing one of these memory cells. This test will verify the correctness of the virtual machine for several of reduction rules given in Table 2.3.

Setup

The memory cell example was coded as a π-Calculus program and used for this test. This program is illustrated in Section A.1 of the appendices. The program when given to the Stand-Alone machine should reduce to the following two processes, where $helloworld$ is the value that was temporary stored at the cell.

$$*createcell?(value, getcell).(\#cell)(cell![value] \mid getcell![cell])$$
$$\mid \quad helloworld![\,]$$

Results

At the end of the development, the results obtained where satisfying. The expected results where obtained. This test served for the verification of the rules **r-communication** and **r-replicated-communication** of Table 2.3. We also tested the scoping principle, and verified that bound names and free names are handled correctly.

6.1.2 Test case - Changing the network of communication

We next test the machine using the π-Calculus program given in Section A.2. This program illustrates how the π-Calculus capabilities of how the communication network between processes change dynamically. Let us first examine the logical meaning behind this program. Take the following system of processes.

$$P \mid Q \mid R$$

$$
\begin{aligned}
^{where}\ P &\implies one?(channel).\text{if } channel = change \text{ then } ((\#n)one![n].n?(message)) \\
Q &\implies one![change].one?(new).two![new] \\
R &\implies two?(new).new![helloworld]
\end{aligned}
$$

As one can note process P is unaware of process R and these cannot communicate. However, process Q will communicate with process P the channel $change$. Hence, the process P will check for the condition $channel = change$. If this is so, then the process P communicates a fresh channel name to the process Q which will in turn communicate with process R. Therefore, at this stage, process R is aware of process P by means of the new channel.

Setup

The Stand-Alone virtual machine is given the program as illustrated in Section A.2. The machine will perform a correct simulation if the end results consists of inactive processes, meaning that communication has been achieved correctly throughout.

Results

At this stage some problems where identified, since the virtual machine was not producing correct results as expected. However, these problems where tackled and the tests where performed once again. Variations to this test where used to help in the debugging of the problem. Finally we have satisfying results on the outcome.

6.1.3 Remarks

Throughout the evaluation stages of the Stand-Alone virtual machine, we note a number of faults. The most involving problems that we needed to solve where the ones related to the rule **r-replicated-communication** and the handling of bound names. The utmost of efforts have been done to find a solution to these problems, and which was finally found. It is important to mention that, the testing during this stage served as an important phase to fully understand the concepts of the π-Calculus notation.

6.2 Testing the Interactive Virtual Machine

The Interactive Virtual Machine will be mainly tested for its interactive features. Note that the reduction procedures that are implemented within the Interactive virtual machine, are identical to the reduction procedures within the Stand-Alone virtual machine. Thus, the Interactive machine was tested using, similar tests as the ones performed on the Stand-Alone machine. This ensured us that the internal operations are consistent between the two virtual machines. Therefore, the testing that is performed here, is to verify that the user-to-machine interaction is achieving accurate results. We then test the machine for the feature of abstraction that it offers, by constructing a π-Calculus program with similar functions as the one used during the first test, and investigate on whether the user can notice the difference between the two programs.

6.2.1 Test case - Stack A

A stack implementation was constructed using π-Calculus notation, and this was then imported to a Π-Language program as illustrated in Section A.3. The Stack program

provides a π-Calculus declaration, which when invoked, this will return a unique channel to a stack, (b). This channel (b) representing a stack, will then return two channels, on for pushing channel names on top of the stack ($push$), and another to pop channel names from the top of the stack (pop). When channel names are outputted using the push, then this represents the push feature. On the other hand, to pop channel name, we first have to output a listener channel, and then we pop the channel name to the top of the stack, by inputting from this listener channel.

Setup

Figure 6.2 gives a trace through the steps involved during the test. We load the stack program to the Interactive machine at **Step 1**. This gives us the channel *stack*, indicating that an input is available, hence in **Step 2** we output on the channel *stack* a channel name *mystack*. The channel *mystack* will then indicate the availability of output. Note as well that the types of the channels are changing accordingly. These types are the same as declared in the program and these can be viewed by the user during interaction. At **Step 3**, we input from the channel *mystack* two values, and we use the variables *mypush* and *mypop*. Notice how these variables are given the corresponding channel names, and how these channel names are added to our table. At this stage we can start pushing data on to the stack. At **Step 4** we push the channel names *one*, *two* and *three*

We now test that the stack is performing the expected job, by performing a pop. Thus, at **Step 5** we output the channel *listener* on the channel *pop0*. The channel *listener* is correctly added to the table, indicating that it is ready to output a channel name. Therefore, at **Step 6** we perform an input from the channel *listener*. The variable *value* is assigned the channel name *three*, which is the expected result since the channel name *three*, was the last name that was pushed on the stack.

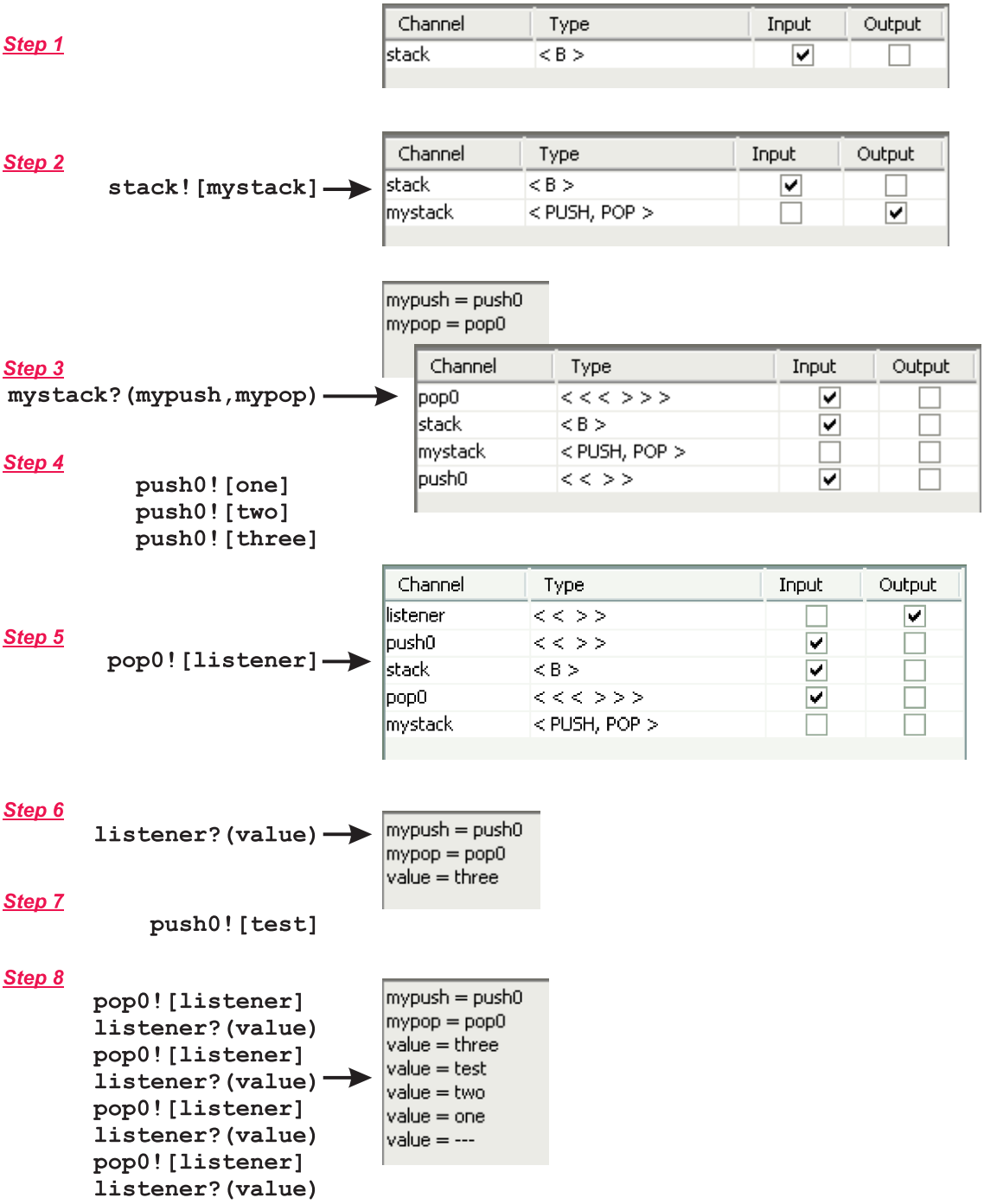


Figure 6.2: Trace of stack program

To confirm that the stack is working accurately, at **Step 7** we push the channel name *test* on to the stack. Hence, at **Step 8** we empty the stack by repeating the steps 5 and 6 for each value to be popped from the stack. As one can notice, the returning results are as expected, meaning that the first channel name retrieved was *test*, after which the names *two* and *one*, where retrieved sequentially. Notice, how the last pop did not retrieve any channel name, since the stack is empty.

Results

The expected results have were achieved from the test that was applied. Variations to the sequence given in Figure 6.2, where performed as a test to verify that the Interactive virtual machine is simulating the execution of π-Calculus correctly. These tests and the one discussed previously, gave very promising results. Note how program Stack A in Section A.3 is testing all of the features tackled throughout this dissertation. We even verified that the typing column is showing accurate values, meaning that the type given to the channel is the correct one.

6.2.2 Test case - Program Details Abstraction

After that the Interactive virtual machine has been verified as correct and that it is producing accurate results as expected, we investigated on the interactive feature of the machine. We developed a π-Calculus program with the same functionalities as that of Stack A that was used for the previous test. This program is given in Section A.4 of the appendices. The difference in this stack implantation (Stack B) is that the channel names are being stored using two lists rather the a single list as in Stack A. This means that the head of the stack will iterate the push feature between the two stacks, and for the pop feature the same principle applies but it has to be done in an opposite direction.

Setup

The plan for this test was to perform the same sequence of procedures, as performed in the previous test on both stack implementations. We simply switch the Stack A program with that of Stack B, and perform the same test. This would verify that the two programs have consistent functionalities, thus the user is unable to distinguish the Stack A program from the Stack B program.

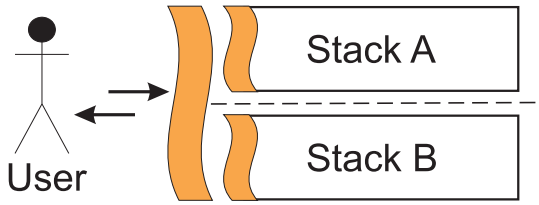


Figure 6.3: Program details are abstracted away from the user

Results

This test turned out to be most satisfying, since no difference could be noted between the two implementations. Both stacks are instantiated in the same manner, and both provide the *push* channel and the *pop* channel. Other tests have been performed, and it was noted that the two programs have to be constructed using the same "interface", meaning that the interaction with the user has to provide the same type of channels and amount.

6.2.3 Remarks

Promising results have been achieved through these tests that have been carried out. To a certain point we proved the correctness of the machines, and verified most of our specula-

tions especially the interactive abstraction that the Interactive virtual machine is capable of offering.

6.3 Limitations

Following the evaluation stage, we can discuss a number of known problems and the limitations of the application that has been developed.

Both of our virtual machines do not conduct any *Garbage Collection*. By Garbage Collection we understand, that those channels that are no longer referenced, or no longer used, then these can be disposed. By disposing of these channels, the machine will be able to perform more efficiently since, the machine will be accounting for resource handling.

Another limitation that our machine has is the lack of the Summation operator. This was already discussed earlier, and we explained how this was purposely omitted, because it offers more complexity to the overall structure of the machines. However, it still remains listed as a limitation since certain π-Calculus program which makes use of this operator cannot be simulated on our machines.

A crucial limitation that was noted concerns channel typing. Since, the typing mechanism that was adopted for the development of the virtual machines is a limited one, then this also limits the capabilities of the virtual machines. We identify the Interactive Virtual Machine more restricted in this significance, since the Stand-Alone machine can be executed without the Type-Checker module, thus allowing it to simulate even complex programs.

Chapter 7

Conclusions and Future Work

Throughout this dissertation we have seen how π-Calculus notation was easily encapsulated into a simple programming language, such as the Π-Language. For this dissertation, the Π-Language described in this study, has proven to be a decent language, to program a π-Calculus program. We even described a simple typing system to be used when composing π-Calculus programs, and this was followed by a Type-Checker to ensure correct channel typing. We then described the construction of the intermediate code representation for π-Calculus. We suggested that this representation would act as the transitional point between π-Calculus notation and various virtual machines. This guided us to develop a compiler to translate π-Calculus notation into this intermediate representation. We then focused on Turner's, Abstract Machine for π-Calculus, and we illustrate how a similar machine can be implemented. This virtual machine was able to interpret π-Calculus programs, and simulate the communication of the processes within an encloses environment. We next suggested to develop an Interactive virtual machine, which will extend the functionality of the Stand-Alone virtual machine, by allowing an external source to communicate with the machine. We even proposed, how such a machine would be capable of abstracting the implementation details of π-Calculus programs from the user.

The virtual machine implementations where then evaluated and tested for correctness and accuracy. A number of examples where introduced, and these were used to verify that the Stand-Alone virtual machine is able to produce correct results, when compared with the realistic counterparts. Subsequently, the Interactive virtual machine was tested using these examples. However, special interest was given to the two different Stack implementations. These π-Calculus programs offered a stack data structure, using the same user-to-machine interface, but have a different inner implementation. We proved that the Interactive virtual machine, does indeed provide a layer of abstraction to the user. The results demonstrated how, a typical user is unable to distinguish between two π-Calculus programs, that offer the same functionalities but with different internal methodologies.

At this point we can suggest a few ideas for possible further work. Maybe the most obvious task to follow would be to improve on the limitations of the developed virtual machines. As we have stated, the current implementations to not offer Garbage Collection for unreferenced channels and it does not offer the possibility to use the Summation operator. Hence, a potential task would be to develop a virtual machine which extends these functionalities.

Throughout this study we did not focus much on typing mechanisms, since this was beyond the initial objectives. However, we did tackle simple channel typing in brief, and this has an interesting topic in the field of π-Calculus. For that reason, a prospective task would be to elaborate on typing techniques for π-Calculus. We believe that typing is essential to carry out accurate interactivity, between the user and the Interactive virtual machine. A potential project would be to develop a system of interactive machine, which will act as interactive agents over a distributed system.

In conclusion we believe that, the study about π-Calculus has served as a stepping stone for observing a different perspective of programming, and appreciating detailed issues regarding concurrency and communication, from a distinct point of view.

Appendix A

Examples of π-Calculus Programs

The following π-Calculus programs were constructed during this study, as part of the objectives for the dissertation. These examples are used in Chapter 6, during the testing phase of the development.

A.1 Memory cell

```
ch createcell := < < >, < < < > > > >;
ch cell := < < > >;
ch helloworld := < >;
ch listener := < < < > > >;

begin
  *createcell?(value,getcell).(#cell)(cell![value] | getcell![cell])
  | createcell![helloworld,listener]
          .listener?(myfirstcell)
          .myfirstcell?(message)
          .message![]
end
```

A.2 Changing the network of communication

```
var COM := < < < > > >;
ch one := COM;
ch two := COM;
ch change := < < > >;
ch n := < < > >;
ch helloworld := < >;

begin
  one?(channel).if channel=change then ((#n)one![n].n?(message))
  | one![change].one?(new).two![new]
  | two?(new).new![helloworld]
end
```

A.3 Stack A

```
var PUSH := < < > >;
var POP := < < < > > >;
var B := <PUSH, POP>;
ch stack := <B>;
ch b := B;
ch pop := POP;
ch push := PUSH;
var rec CELL := < < >, CELL>;
ch a := CELL;
ch endd := CELL;
ch head := <CELL>;
ch createcell := < < >, CELL, <CELL> >;

def stack(b)
begin
  (#endd,head,createcell,push,pop)
  (
     b![push,pop].
     (
        head![endd].0
        | *createcell?(x,next,ret).(#a)(a![x,next].0 | ret![a].0)
        | *push?(x).head?(y).createcell![x,y,head].0
        | *pop?(z).head?(x).if x = endd
                           then (head![endd].0)
                           else (x?(v,w).(z![v].0 | head![w].0))
     )
  )
end
```

A.4 Stack B

```
var PUSH := < < > >;
var POP := < < < > > >;
var B := <PUSH, POP>;
ch stack := <B>;
ch b := B;
ch pop := POP;
ch push := PUSH;
var rec CELL := < < >, CELL>;
ch a := CELL;
ch end1 := CELL;
ch head1 := <CELL>;
ch end2 := CELL;
ch head2 := <CELL>;
ch current := < <CELL> >;
ch createcell := < < >, CELL, <CELL> >;
ch reset := < >;

def stack(b)
begin
  (#end1,end2,head1,head2,createcell,push,pop,current,reset)
  ( b![push,pop].  (
    *reset?().(head1![end1].0 | head2![end2].0 | current![head1].0)
    | reset![]
    | *createcell?(x,next,ret).(#a)(a![x,next].0 | ret![a].0)
    | *push?(x).current?(head).head?(y).createcell![x,y,head].
         if head=head1 then (current![head2]) else (current![head1])
    | *pop?(z).current?(head).
         if head=head1
         then (current![head2].head2?(x).
               if x=end2
               then (reset![]) else (x?(v,w).( z![v] | head2![w])))
         else (current![head1].head1?(x).
               if x=end1
               then (reset![])
               else (x?(v,w).(z![v] | head1![w] ) ) )
  ) )
end
```

Appendix B

EBNF for the Π-Language

The following EBNF describes the language that we designed, including all the required changes. The standard *Extended Backus-Naur form (EBNF)* [ISO96] is used, which is an extension of the basic Backus-Naur form (BNF) meta-syntax notation.

$$
\begin{aligned}
Start &:= (Include)* \ (Declaration)* \ (Definition)* \ (Declaration)* \\
&\quad (< BEGIN > Pipe < END >)? \ < EOF > \\
Include &:= < INCLUDE > File < SEMICOLON > \\
File &:= < STRING > \\
Definition &:= < DEF > DefName < BEGIN > (Declaration)* \\
&\quad Pipe < END > \\
DefName &:= Name < LPAREN > (Tuples)? \ < RPAREN > \\
RecArguments &:= (TypeVariable \mid RecArg \mid Name) \\
&\quad (< COMMA > (TypeVariable \mid RecArg \mid Name))*
\end{aligned}
$$

$$
\begin{aligned}
Arguments &:= (TypeVariable \mid Arg) < COMMA > (TypeVariable \mid Arg)) * \\
Arg &:= \text{" } < \text{"}(Arguments)?\text{" } > \text{"} \\
RecArg &:= \text{" } < \text{"}(RecArguments)?\text{" } > \text{"} \\
TypeVariable &:= < TYPEVAR > \\
DecChannel &:= < CH > Name < ASS > (TypeVariable \mid Arg) \\
&\quad < SEMICOLON > \\
DecVariable &:= < VAR > TypeVariable < ASS > (TypeVariable \mid Arg) \\
&\quad < SEMICOLON > \\
DecRecVariable &:= < VAR >< REC > TypeVariable < ASS > \\
&\quad (TypeVariable \mid RecArg) < SEMICOLON > \\
Declaration &:= (DecChannel \mid DecVariable \mid DecRecVariable) \\
Int &:= (< INTEGER > \mid < STOP >) \\
Pipe &:= (Process(< PIPE > Pipe)?) \mid (Brackets(< PIPE > Pipe)?) \\
Brackets &:= < LPAREN > Pipe < RPAREN > \\
Process &:= Continuation \mid Restriction \\
Continuation &:= (((Input \mid Output \mid Tau \mid ReplicatedInput \mid DefinitionCall \mid \\
&\quad Print)(< CONTINUATION > Process)?) \\
&\quad \mid (IfThen \mid Stop \mid Brackets)) \\
ReplicatedInput &:= < REPLICATION > Input \\
Stop &:= < STOP > \\
Input &:= Channel < INPUT > \text{"("}(NameTuples)?\text{")"} \\
Output &:= Channel < OUTPUT > \text{"["}(Tuples)?\text{"]"} \\
Channel &:= < STRING > \\
Tuples &:= Channel(< COMMA > Channel) * \\
NameTuples &:= Name(< COMMA > Name) * \\
Name &:= < STRING > \\
Tau &:= < TAU > \\
Restriction &:= < LPAREN >< HASH > Channel\ BNames \\
BNames &:= ((< COMMA > Channel\ BNames) \\
&\quad \mid (< RPAREN > (Brackets \mid Process))) \\
IfThen &:= < IF > Expression < THEN > Brackets(< ELSE > Brackets)? \\
Expression &:= Name < EQUALS > Name \\
DefinitionCall &:= Channel < LPAREN > (Tuples)? < RPAREN > \\
Print &:= < PRINT >< LPAREN > Tuples < RPAREN >
\end{aligned}
$$

Appendix C

Class Diagrams

This appendix contains classes diagrams, which describe the most important modules of the application.

The *PiParserVisitor* class represents the visitor interface generated by the JavaCC tools. Several modules implement this interface, such as the *TypeChecker* and the *Compiler*.

The *TaskManager* class is the core module of the Stand-Alone virtual machine. It encapsulates all of the reduction procedures and it handles the process communication.

The *Channel* class represents the channel of communication within the Interactive virtual machine. The main changes that where applied to this machine revolve around the Channel class.

The *Machine* class is the focal point of the Interactive virtual machine, by scheduling the processes on Channel objects. It also acts as the bridge between the user and the internal program, since it provides an interactive interface.

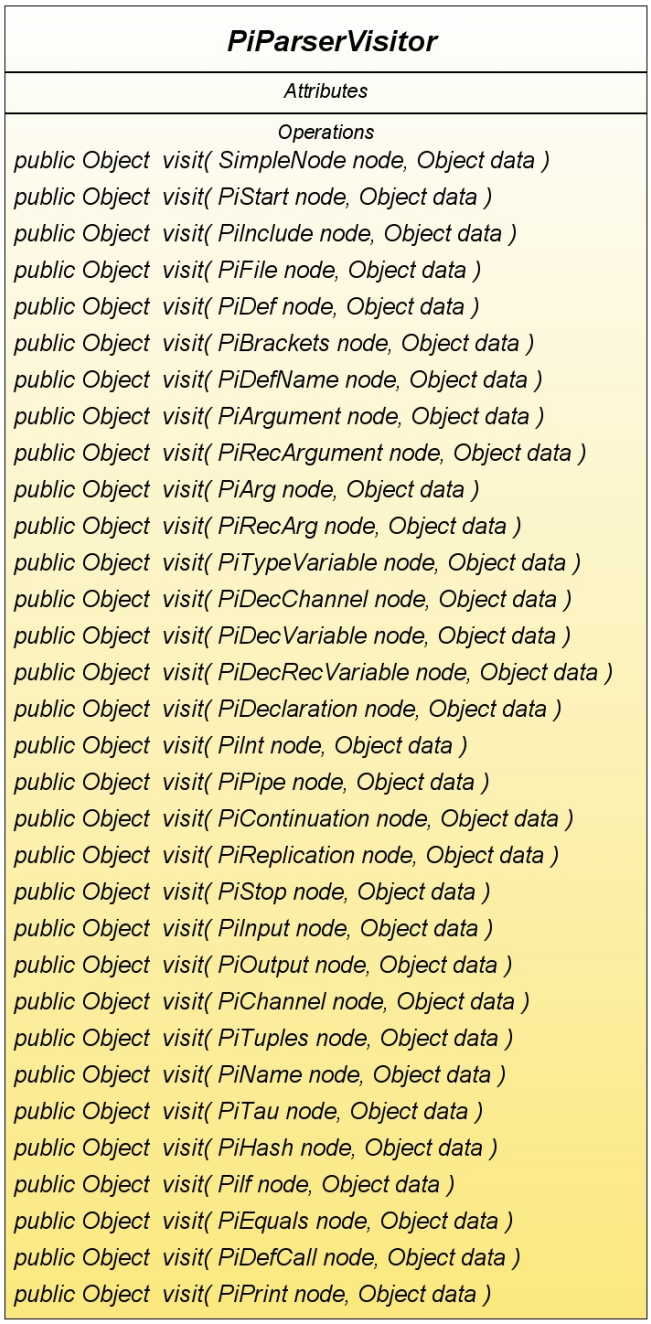


Figure C.1: PiParserVisitor Class Diagram *(Visitor interface generated by JavaCC)*

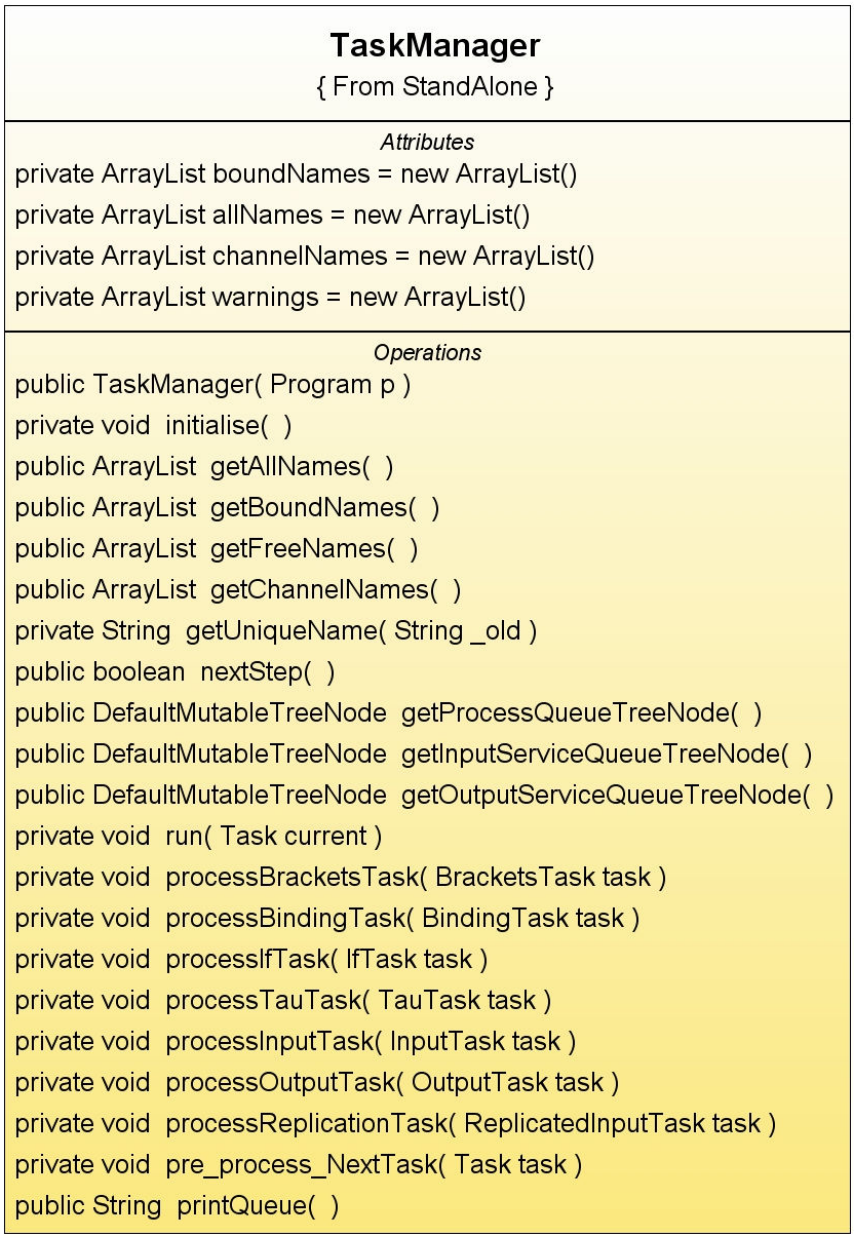


Figure C.2: TaskManager Class Diagram

Figure C.3: Channel Class Diagram

Figure C.4: Machine Class Diagram

Appendix D

User Manual

Figure D.2, gives a screen shot of the application while running in editing mode. One can edit a number of π-Calculus programs. By using the toolbar functions or the main menu, one can create new files, save files, open existing files, and perform the standard functions offered by editors.

Figure D.1: The editor's toolbar

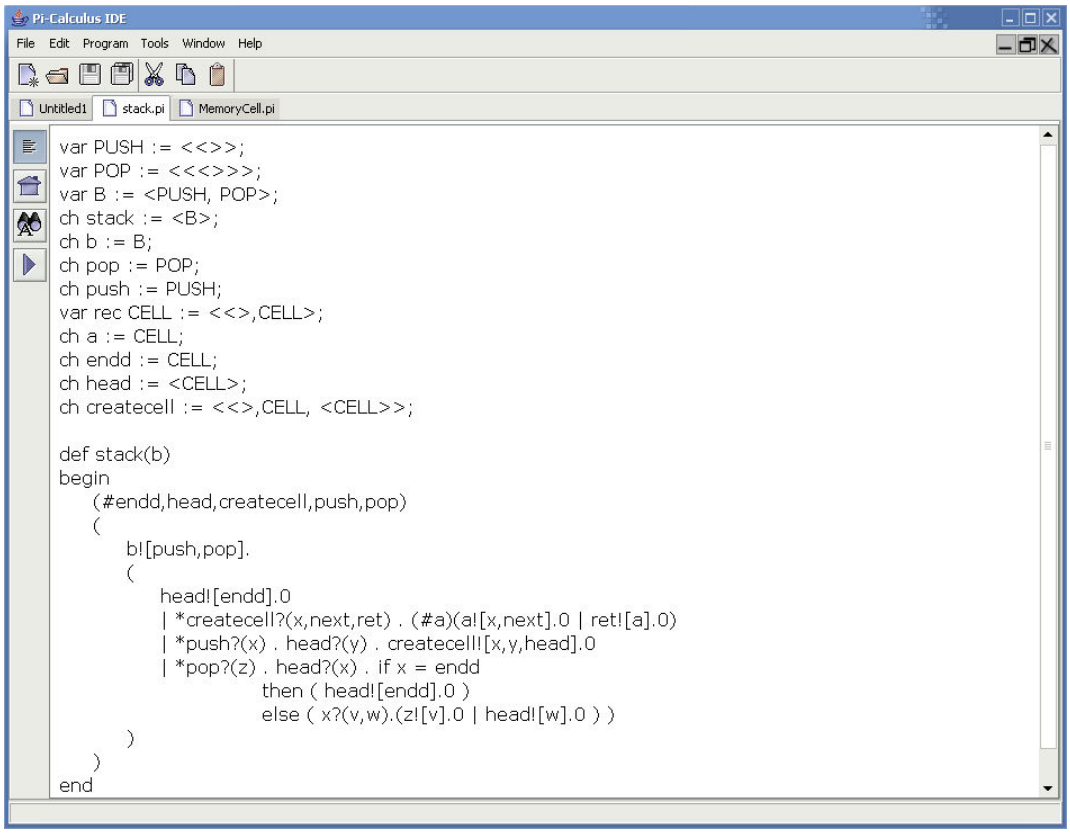


Figure D.2: Application running in editing mode

The application offers MDI support as shown in Figure D.3. Multiple documents can be opened concurrently, either displayed as separate windows or as a serious of tabs.

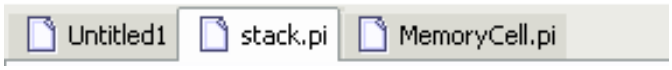


Figure D.3: MDI support

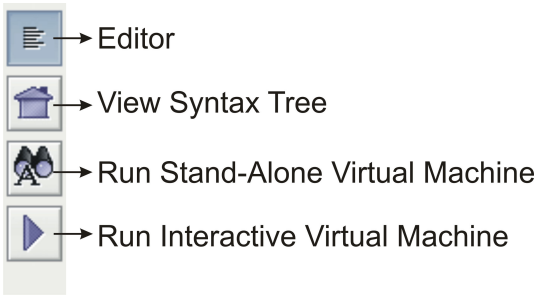


Figure D.4: The editor's side-bar

The editor's side-bar is used to run the three main modules found in the application. One can either parse the program, and display a graphical view of the syntax tree, or simulate the program on the Stand-Alone machine or simulate the program on the Interactive machine.

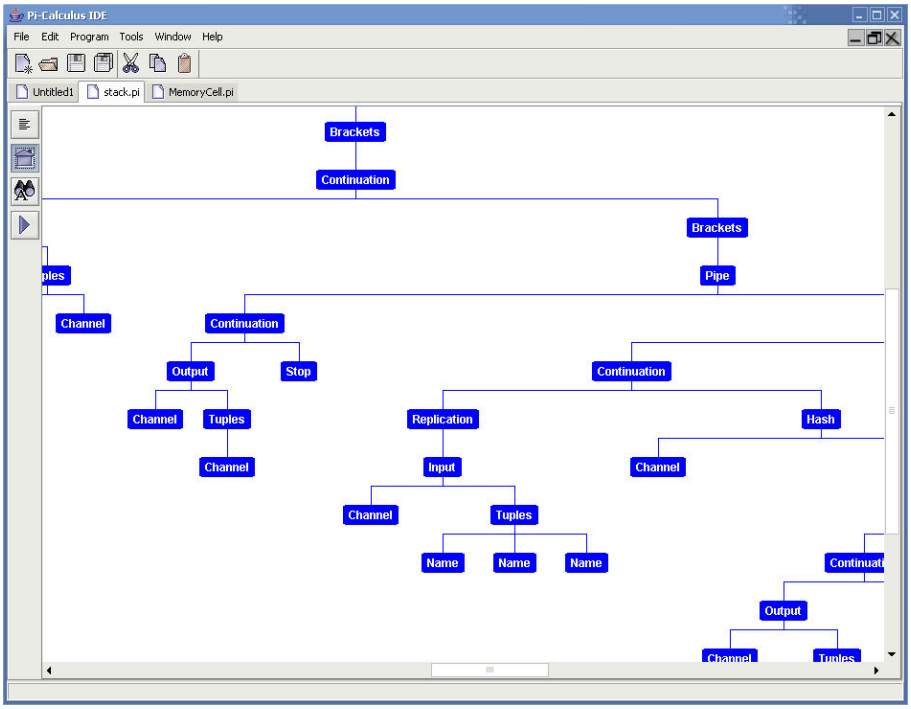


Figure D.5: Showing the Syntax Tree for the program

The user interface for the Stand-Alone machine, illustrates how the machine has computed the program internally.

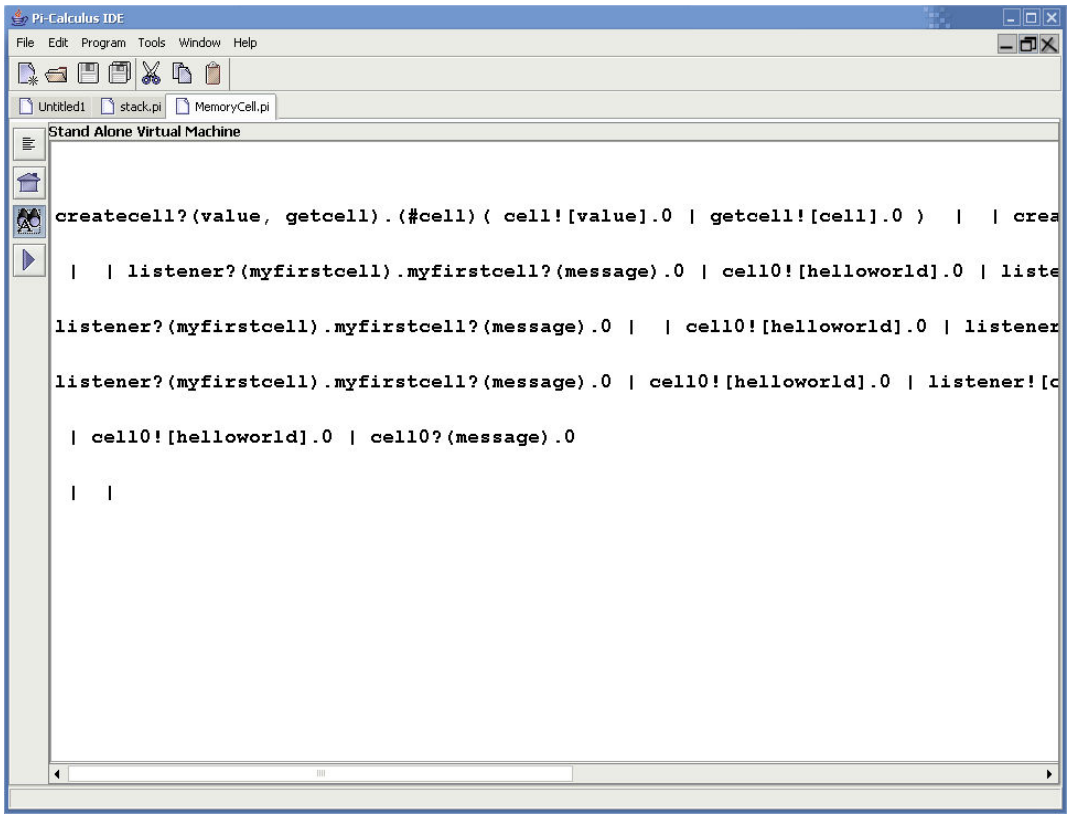


Figure D.6: Running the Stand-Alone virtual machine

The interactive interface offers a panel for the processing of tasks. The user is able to select a task and a channel, and input either a number of names or variables accordingly. For multiple names, or variables these have to be separated by commas. The table show the current state of the Virtual Machine, given a list of channels, their type and the availability for input and output. The bottom panel gives the default output stream that is used to receive channel names.

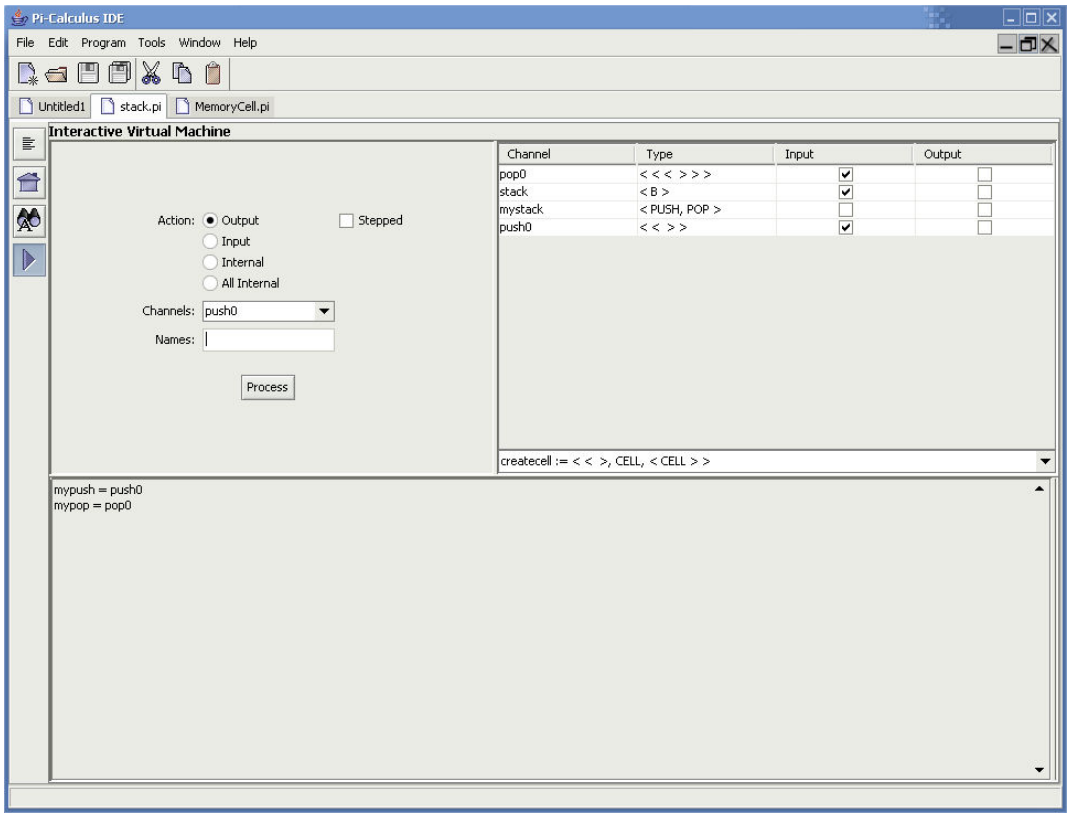
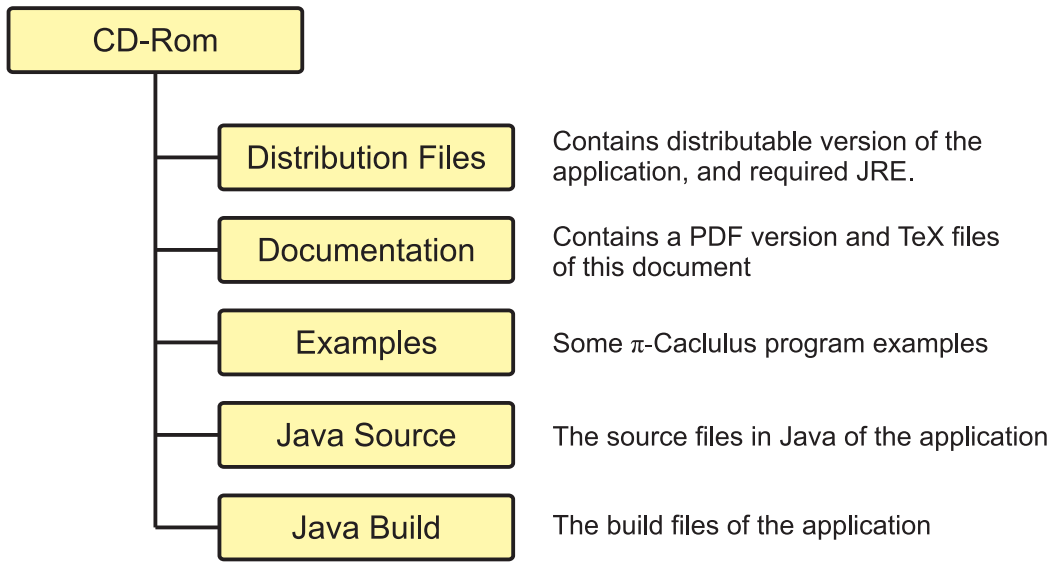


Figure D.7: Running the Interactive virtual machine

Appendix E

Contents of the CD-Rom



Bibliography

[App02] Andrew W. Appel. *Modern Compiler Implementation In Java*. Cambridge Univerity Press, 2002.

[ISO96] ISO/IEC-14977. *Information Technology, Syntactic Metalanguage, Extended BNF*. ISO Standards, 1996.

[Kob] Naoki Kobayashi. Type systems for concurrent programs. Technical report, Department of Computer Science, Tokyo Institute of Technology.

[Mil89a] Robin Milner. A calculus of mobile processes, part i. Technical report, University of Edinburgh, 1989.

[Mil89b] Robin Milner. A calculus of mobile processes, part ii. Technical report, University of Edinburgh, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[PSP98] Pawel T. Wojciechowski Peter Sewell and Benjamin C. Pierce. Location independence for mobile agents. Technical report, In Workshop on Internet Programming Languages, Chicago, May 1998.

[PSP99] Pawel T. Wojciechowski Peter Sewell and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical report, Computer Laboratory, University of Cambridge, 1999.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[Sew00] Peter Sewell. Applied pi - a brief tutorial. Technical report, Computer Laboratory, University of Cambridge, 2000.

[SW] Peter Sewell and Pawel T. Wojciechowski. Nomadic pict: Language and infrastructure design for mobile agents. Technical report, Computer Laboratory, University of Cambridge.

[SW03a] Davide Sangiorgi and David Walker. *The Pi-Calculus : A Theory of Mobile Processes*, chapter 1. 2003.

[SW03b] Davide Sangiorgi and David Walker. *The Pi-Calculus : A Theory of Mobile Processes*, chapter 6. 2003.

[Tur95] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[Wis01] Lucian Wischik. Explicit fusions: Theory and implementation. Technical report, Computer Laboratory, University of Cambridge, 2001.