Analysis of the reachability problem in fragments of the Pi-calculus

Luis Pino

Undergraduate Thesis Report

Analysis of the reachability problem in fragments of the π-calculus

Luis Fernando Pino Duque

Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Santiago de Cali
2010

Undergraduate Thesis Report

Analysis of the reachability problem in fragments of the π-calculus

Luis Fernando Pino Duque

Trabajo de Grado para optar por
el titulo de Ingeniero de Sistemas

Supervisor
Juan Francisco Díaz Frias, Ph.D.
Profesor
Escuela de Ingeniería de Sistemas y Computación
Universidad del Valle

Co-Supervisor
Frank D. Valencia, Ph.D.
Científico Investigador
Laboratoire d'Informatique (LIX)
École Polytechnique de Paris

Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
Santiago de Cali
2010

Nota de Aceptación

_____

_____

_____

_____

_____

_____

Juan Francisco Díaz Frias
Director del Proyecto

_____

Jurado

_____

Jurado

Santiago de Cali, _____

Abstract

The π-calculus is one of the most important formalisms for analyzing and modelling concurrent systems. It is a simple but powerful tool for specifying and checking several properties in this kind of systems. An interesting property of any system is the ability to reach some special state where it has a particular behavior. In security systems this is extremely important, since we would like that a system does not reach a state where a secret becomes observable to potential attackers.

This work studies the reachability problem in fragments of the π-calculus and explores some expressiveness results beyond this problem. We prove the relation between local names and sequences of actions in $\text{CCS}_!$ processes. Using this result and the decidability of barbs from [BGZ09] we prove that the reachability problem for some fragments of π-calculus is decidable. We also provide an algorithmic approach for solving this problem using the theory of well-structured transition systems [FS01], in consequence we are able to verify this property in infinite state systems with a finite number of steps. Finally, we provide a small interpreter for $\text{CCS}_!$, useful as an initial practical approach for checking properties in real life systems specified by this calculus.

Contents

List of Figures

Chapter 1

Introduction

In today's world, technology is one of the most important cores for the development of the society. This role has made access to high-tech devices increasingly frequent. Elements like Web, wireless networks, high capacity laptops, mobile devices, among others, have allowed advance towards information globalization, but carrying with it new challenges and problems that need to be solved for assuring a reliable, correct and secure service.

Computer science offers a framework in which information technology can be formalized, allowing establishing conditions where they work correctly. There are many factors that affect the correctness of such technologies, one of the most recent and challenging is concurrency, and this consists in many processes making use of a system in a simultaneous way. It is here where an area of computer science named concurrency theory goes into action.

In this theory, process calculi are distinguished, its intention is to model and reason about concurrent systems. Such calculi are capable to express systems formally, hence it is possible to argue about them for obtaining correct results. There are many examples such as CCS[1], π-calculus[2], Spi-calculus (π-calculus for arguing about security), among others, they have been specialized for solving specific problems since all of them count on with a modeling approach and an associated expressiveness (a measure of how powerful the calculus is). This analysis is focused on a fragment of the π-calculus that is fundamental in concurrency theory.

This project aims at analyzing the reachability problem in fragments of the π-calculus, such analysis will allow determining if a state of a system can be reached. It also aims at providing an algorithm resulting from the previous step, and for

[1] Calculus of Communicating Systems more information in:
http://en.wikipedia.org/wiki/Calculus_of_communicating_systems

[2] More information in: http://en.wikipedia.org/wiki/Pi-calculus

practical effects, another analysis focused on the complexity of such algorithm.

1.1 Problem Description

Concurrency theory investigates how to analyze those systems where many processes act in a simultaneous way, and arguing about them for obtaining conclusions about correctness, security, reliability and other important aspects.

Process calculi are used for this purpose, because they allow making process modeling in concurrent systems, and depending of its specialization they are able to express, until certain point, a series of actions that can be object of study for a subsequent reasoning.

The π-calculus is one of the most influential calculi in concurrency theory, its simplicity and powerful operators make it an excellent tool for modeling concurrent systems. It also allows expressing mobile behavior and it is widely used around the world for analyzing several kinds of systems. This project focuses in this calculus due to its importance and by the impact of the results which could be easily used worldwide.

Nowadays, the notion of security has become one of the most important topics in research. This is because we have very complex systems and they represent a giant pillar in every second of our daily life. Moreover, in most cases the information must be managed carefully and privacy (among other aspects) is hugely valuable. Then, making a system safe means that it cannot reach a state in which private information can be revealed to an external agent that should not know such information.

By these reasons, we can say that establishing which are the states that make the system to be in danger are very important for arguing about its security. This problem is called *reachability problem*, and it is a central topic in security research.

Therefore, the problem is to determine whether a specific action β can be reached in some execution of the system. Such system is specified using the summation-free zero-adic fragment of the π-calculus ($\text{CCS}_!$). This project will provide a decidability analysis of this problem by proving (or refusing) the strict relationship between the actions needed to reach such state and the use of local names.

The next step will be to bring the theory into practice, by giving an algorithm that can determine (given a process P and an action β) whether such action can be reached by using the information proved previously. For practical purposes, the

complexity of that algorithm is very important, and then it will conclude the analysis proposed in this project.

1.2 Objectives

1.2.1 Main Objective

To analyze the reachability problem in the $CCS_!$.

1.2.2 Specific Objectives

- To determine the decidability of the reachability problem for the $CCS_!$.
- To propose an algorithm for determining the reachability of an action given a process written in the $CCS_!$.
- To analyze the complexity of the proposed algorithm.

1.3 Justification

For long time, sequential systems have been the center of computer science studies, but these approaches have not been enough for a growing world that demands more complex systems everyday. In that sense, concurrent systems are present in our daily activities, from our cells to social networks. In the late 20^{th} and 21^{st} century the core of our society is based on information systems that carry out all kind of tasks.

Following this idea, analyzing concurrent systems must be a central activity in computer science. Here is where the π-calculus goes into scene, because it is an excellent tool for modeling concurrent and mobile systems, for this reason it is one the most influential process calculus in concurrency theory.

Another important thing is that system's security is a growing problem that has always been present in computer science, but in the last decade has become a very important topic in research. The reason of its importance lies in the increasing complexity of systems, then it is difficult to determine the "danger" states and to predict the behavior of them. That is why it is important to have theoretical tools that allow facing this problem.

The importance of this project lies in the reinforcement of the π-calculus by giving an analysis that could be very important for modeling and evaluating systems.

The main advantage will be given in security analysis which is a essential topic for computer science research.

As an additional benefit, all the results obtained here can be linked with the work made by Aranda[Ara09] in his Ph.D. thesis about the expressiveness in π-calculus. That work has inspired the fundamental aspects of this project, then our analysis will allow proving other interesting things about expressiveness due to its relation with such work.

1.4 Background

1.4.1 Technical Background

Process Calculi

The process calculi are a diverse family of related approaches to formally modeling concurrent systems. Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes.

There are many different process calculi in the literature mainly agreeing in their emphasis upon algebra. The main representatives are CCS[Mil89], CSP[Hoa85] and the process algebra ACP[BK85, BW90]. The distinctions among these calculi arise from issues such as the process constructions considered (i.e., the language of processes), the methods used for giving meaning to process terms (i.e. the semantics), and the methods to reason about process behavior (e.g., process equivalences or process logics). Some other issues addressed in the theory of these calculi are their expressive power, and analysis of their behavioral equivalences. We will describe some of the issues named previously.

The π-calculus

Syntax. *Names* are the most primitive entities in the π-calculus. We presuppose a countable set $\mathcal{N}$ of (port, links or channel) *names*, ranged over by $x, y, \ldots$. For each name x, we assume a *co-name* $\overline{x}$ thought of as *complementary*, so we decree that $\overline{\overline{x}} = x$. We use $\vec{x}$ to denote a finite sequence of names $x_1 x_2 \ldots x_n$. The other entity in the π-calculus is a *process.* Processes are built from names as follows.

Definition 1.4.1. *(Syntax) The processes, the summations and the prefixes in π-*

calculus are given respectively by:

$$\begin{aligned} P &:= M \mid (\nu x)P \mid P \mid P \mid !P \\ M &:= 0 \mid \pi.P \mid M + M' \\ \pi &:= x(y) \mid \overline{x}y \mid \tau \end{aligned}$$

First we explain the summations and the prefixes and then the processes. The process (summation) 0 does nothing. $\overline{x}y.P$ and $x(y).P$ represent the output and input process respectively, $\overline{x}y.P$ is a process which can output a datum y on channel x and then it behaves like P, $\overline{x}y$ is called a guard or *(output) prefix*. $x(y).P$ is a process which can perform an input action on channel x and then it behaves like $P\{z/y\}$, the process which has replaced every occurrence of the name y, by the datum z received, $\{z/y\}$ is a substitution of z by y, $x(y)$ is called a guard or *(input) prefix*. $\tau.P$ can evolve invisibly to P. τ can be thought of as expressing an internal action of a process, τ is called a guard or *(unobservable) prefix*. the sum (or choice) $P + Q$ represents the process which can has the capabilities of either P or Q but not both. Once a capability of P (Q) has been performed, Q (respectively, P) is disregarded.

In $P \mid Q$, the parallel composition of P and Q, P and Q can proceed independently or can synchronise via shared names. In $(\nu x)P$, the name x is declared private to P, i.e. initially, components of P can use x tinteract with one another but not with other processes, the scope of x could change as a result of interaction between processes as will be seen later. Finally, the replication $!P$ can be thought of as unboundedly many P's in parallel $P \mid P \mid P \mid \ldots$, replication is the means to express infinite behaviour. Notice that the operands in a sum must themselves be summations. Hence it says that the π-calculus considers guarded-choice.

In each of $x(y).P$ and $(\nu y)P$, the occurrence of y is *binding* with *scope* P. An occurrence of a name in a process is *bound* if it is under the scope of a binding occurrence of the name. An occurrence of a name is *free* if it is not bound. Given Q we define its *bound names* $bn(Q)$ as the set of names with a bound occurrence in Q, and its *free names* $fn(Q)$ as the set of names with a non-bound occurrence in Q, hence $n(Q) = fn(Q) \cup bn(Q)$ is the set of names of Q.

As consequence of the interchange of names between processes an unintended capture of names by binders could arise, to avoid it, the following definition of α-convertability is useful.

Definition 1.4.2. *(α-convertability) [SW01]*

1. *If the name w does not occur in the process P, then $P\{w/z\}$ is the process obtained by replacing each occurence of z in P by w.*

2. *A change of bound names in a process P is the replacement of a subterm $x(z).P$ of P by $x(w).Q\{w/z\}$, or the replacement of a subterm $(\nu z)Q$ of P by $(\nu w)Q\{w/z\}$, where in each case w does not occur in Q.*

3. *Processes P and Q are α-convertible, $P = Q$, if Q can be obtained from P by a finite number of changes of bound names.*

Hence we adopt two well-known conventions:

Convention 1.4.1. *[SW01] Processes that are α-convertible are identified.*

Convention 1.4.2. *[SW01] When considering a collection of processes and substitutions, we assume that the bound names of the processes are chosen to be different from their free names and from the names of the substitutions.*

Semantics. The semantics of the language described above made precise by a labelled transition system. A transition $P \xrightarrow{\alpha} Q$ says that P can perform an *action* α and evolve into Q. The set of actions used in the transition system is composed by $\bar{x}y$, xy, $\bar{x}(y)$, τ. $\bar{x}y$, *a free output*, sends the name y on the name x, xy, *an input*, receives the name y on the name x, $\bar{x}(y)$, *a bound output*, sends a fresh name on x and τ is an *internal action* .

Definition 1.4.3. *(Actions) The actions, which are ranged over by α, are given by:*

$$\alpha \;:=\; \bar{x}y \;\Big|\; xy \;\Big|\; \bar{x}(y) \;\Big|\; \tau \tag{1.1}$$

Act refers to the set of actions. The set of labels, ranged over by l and l', is $\mathcal{L}$ which is composed of all non-internal actions.

Functions $fn(\_)$, $bn(\_)$ and $n(\_)$ are extended to cope with labels as follows:

$$\begin{array}{cccccccccccc} bn(xy) & = & \emptyset & bn(\bar{x}y) & = & \emptyset & bn(\bar{x}(y)) & = & \{y\} & bn(\tau) & = & \emptyset \\ fn(xy) & = & \{x,y\} & fn(\bar{x}y) & = & \{x,y\} & fn(\bar{x}(y)) & = & \{x\} & fn(\tau) & = & \emptyset \end{array}$$

The *subject* , $subj(\_)$, and *object* , $obj(\_)$, of these actions is defined as: $subj(xy) = subj(\bar{x}y) = subj(\bar{x}(y)) = x$, $obj(xy) = obj(\bar{x}y) = obj(\bar{x}(y)) = y$, $subj(\tau) = obj(\tau) = \emptyset$.

Definition 1.4.4. *(Semantics) The labelled transition relation $\xrightarrow{\alpha}$ is given by the rules in Table 1.1. Omited from Table 1.1 are the symmetric forms of Sum-L, Par-L, Com-L and Close-L. Let us define the relation $\stackrel{\alpha}{\Longrightarrow}$, with $s = \alpha_1 \ldots \alpha_n \in Act^*$, as $(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^*$. $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$. $\stackrel{\hat{\tau}}{\Longrightarrow}$ is $\Longrightarrow$ and $\stackrel{\hat{\beta}}{\Longrightarrow}$ is $\stackrel{\beta}{\Longrightarrow}$.*

Some comments on the rules: the side-condition in Rule Par-L rule avoids the capture of a name by the extrusion of the scope of another name. The Open rule expresses extrusion of the scope of a name, this action allows the passing of a name beyond its original scope, its side-condition avoids the execution of an action whose subject is a bound-name as it should not interact with other processes out of the scope of the name. Rule Close-L reflects the interaction between processes in which the left-process has transmitted a bound name to the right-process, thus the scope of the restricted name is extended to include the process which receives it.

$$\text{Input} \quad x(y).P \xrightarrow{xz} P\{z/y\} \text{ where } x,\ y \in \mathcal{N}$$

$$\text{Output} \quad \bar{x}y.P \xrightarrow{\bar{x}y} P \qquad\qquad \text{Tau} \quad \tau.P \xrightarrow{\tau} P$$

$$\text{Sum-L} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\text{Open} \quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} x \neq y \qquad\qquad \text{Res} \quad \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} y \notin n(\alpha)$$

$$\text{Par-L} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} bn(\alpha) \cap fn(Q) = \emptyset$$

$$\text{Com-L} \quad \frac{P \xrightarrow{\bar{x}y} P',\ Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad\qquad \text{Close-L} \quad \frac{P \xrightarrow{\bar{x}(y)} P',\ Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')}$$

$$\text{Rep-Act} \quad \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\text{Rep-Comm} \quad \frac{P \xrightarrow{\bar{x}y} P',\ P \xrightarrow{xy} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

$$\text{Rep-Close} \quad \frac{P \xrightarrow{\bar{x}(z)} P',\ P \xrightarrow{xz} P''}{!P \xrightarrow{\tau} ((\nu z)(P' \mid P'')) \mid !P} z \notin fn(P)$$

Table 1.1: Operational semantics for the π-calculus.

Remark 1.4.1. *We abbreviate, for any names x, y, the guards $x(y)$ and $\bar{x}y$ by x and $\bar{x}$, respectively, where y, is a dummy name: in these cases the datum which can be received or sent is irrelevant*

Notation 1.4.1. *Throughout this work, we use $(\nu a_1 \ldots a_n)P$ as a short hand for $(\nu a_1) \ldots (\nu a_n)P$. We often omit the "0" in $\alpha.0$.*

Now we define $\equiv$ which shall be useful in the project although it is not included in the semantics:

Definition 1.4.5. *Let $\equiv$ be the smallest congruence over processes satisfying α-equivalence, the commutative monoid laws for composition with 0 as identity, the replication law $!P \equiv P|!P$, the restriction laws $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$ and the extrusion law: $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ if $x \notin fn(P)$.*

The Calculus of Communicating Systems

Undoubtedly CCS [Mil89], a calculus for synchronous communication, remains as a standard representative of process calculi. In fact, many foundational ideas in the theory of concurrency have sprung from this calculus. In the following we shall consider two variants of CCS according to its mechanism to model infinite behaviour. Hence, first we show the Finite fragment of CCS and then we introduce the two "recursive" extensions.

Finite CCS. The finite CCS processes can be obtained as a restriction of the finite processes of the π-calculus, i.e those π processes without occurrence of a term of the form $!P$, by requiring all inputs and outputs to have empty subjects only. Intuitively, this means that in CCS there is no sending/receiving of links but synchronisation on them.

In CCS, the actions are names, co-names and τ and therefore, we shall use $l, l', \ldots$ to range over names and co-names, where $\mathcal{L}$ is the set of names and co-names. The set of *actions Act*, ranged over by α and β, extends $\mathcal{L}$ with the symbol τ.

The syntax of finite CCS processes is the following:

Definition 1.4.6. *(Syntax) Processes in finite CCS are given respectively by:*

$$\begin{array}{rcl} P & := & M \mid (\nu x)P \mid P \mid P \mid !P \\ M & := & 0 \mid \pi.P \mid M + M' \\ \pi & := & x \mid \overline{x} \mid \tau \end{array}$$

Definition 1.4.7. *(Semantics) The labelled transition relation $\xrightarrow{\alpha}$ is given by the rules in Table 1.2. Omitted from Table 1.2 are the symmetric forms of Par-L, Com-L and Close-L.*

There are two variants of CCS which extend the above syntax to express infinite behaviour in a different way. We describe them next.

$$\text{Input} \quad x.P \xrightarrow{x} P$$

$$\text{Output} \quad \bar{x}.P \xrightarrow{\bar{x}} P \qquad\qquad \text{Tau} \quad \tau.P \xrightarrow{\tau} P$$

$$\text{Sum-L} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad\qquad \text{Res} \quad \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'}\, y \notin n(\alpha)$$

$$\text{Par-L} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad \text{Com-L} \quad \frac{P \xrightarrow{\bar{x}} P',\ Q \xrightarrow{x} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Table 1.2: Operational semantics for the finite CCS.

Replication $\text{CCS}_!$: As said before, replication is the way of expressing infinite behaviour which has been used in the π-calculus and the Aπ-calculus. It has also studied in the context of CCS in [BGZ09, GSV04]. For replication the syntax of finite processes (Definition 1.4.6) is extended as follows:

$$P, Q, \ldots := \ldots \mid\, !P \tag{1.2}$$

$\text{CCS}_!$ is the restriction of the π-calculus seen by requiring all inputs and outputs to have empty subjects only. The operational rules for $\text{CCS}_!$ are those in Table 1.2 plus the following rules:

$$\text{Rep-Act} \quad \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P} \qquad\qquad \text{Rep-Comm} \quad \frac{P \xrightarrow{\bar{x}} P' \qquad P \xrightarrow{x} P''}{!P \xrightarrow{\tau} P' \mid P'' \mid !P}$$

Table 1.3: Transition Rules for Replication in $\text{CCS}_!$

The study of local names in $\text{CCS}_!$ is central in our work, we introduce the concept of maximum nesting of local names:

Definition 1.4.8. *The maximal number of nesting of local names $|P|_\nu$ can be inductively given as follows:*

$$|(\nu\, x)P|_\nu = 1 + |P|_\nu \qquad |P \mid Q|_\nu = max(|P|_\nu, |Q|_\nu)$$
$$|\alpha.P|_\nu = |!P|_\nu = |P|_\nu \qquad |0|_\nu = 0$$

We also need the concept of weak barb in $\text{CCS}_!$:

Definition 1.4.9 (Weak Barb)**.** *A process P has a barb x denoted by $P \Downarrow x$ iff $P \rightarrow^* \xrightarrow{\alpha}$ and $x \in \alpha$.*

Parametric Definitions: CCS and CCS_p: A typical way of specifying infinite behaviour is by using parametric definitions [Mil99]. In this case we extend the syntax of finite processes (Definition 1.4.6) as follows:

$$P, Q, \ldots := \ldots \mid A(y_1, \ldots, y_n) \tag{1.3}$$

Here $A(y_1, \ldots, y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. We assume that every such an identifier has a unique, possibly recursive, *definition* $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its *body* P_A with each y_i replacing the *formal parameter* x_i. For each $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$, we require $fn(P_A) \subseteq \{x_1, \ldots, x_n\}$.

Following [GSV04], we use CCS_p to denote the calculus with parametric definitions with the above syntactic restrictions.

Remark 1.4.2. *As shown in [GSV04], however, CCS_p is equivalent w.r.t. strong bisimilarity to the standard CCS. We shall then take the liberty of using the terms CCS and CCS_p to denote the calculus with parametric definitions as done in [Mil99].*

The rules for CCS_p are those in Table 1.2 plus the rule:

$$\text{CALL}\ \frac{P_A[y_1, \ldots, y_n/x_1, \ldots, x_n] \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_n) \xrightarrow{\alpha} P'} \ \text{if } A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A \tag{1.4}$$

As usual $P[y_1 \ldots y_n/x_1 \ldots x_n]$ results from replacing every free occurrence of x_i with y_i renaming bound names in P wherever needed to avoid capture.

Notions and equivalences

A central concept is the notion of *encoding* : A map from the terms of a π-calculus variant (e.g., CCS_p) into the terms of another (e.g., $\text{CCS}_!$). The existence of encodings that satisfy certain properties is typically used as a measure of expressiveness (see [CCP06, Gor07, Gor06, CM03, CC01, Pal03]).

Bisimilarity

Definition 1.4.10 (Reduction Bisimilarity)**.** *A reduction simulation is a binary relation $\mathcal{R}$ satisfying the following: $(P, Q) \in \mathcal{R}$ implies that:*

- *if* $P \xrightarrow{\tau} P'$ *then* $\exists Q' : Q \xrightarrow{\tau} Q' \wedge (P', Q') \in \mathcal{R}$.

The relation $\mathcal{R}$ *is a reduction bisimulation iff both* $\mathcal{R}$ *and its converse* $\mathcal{R}^{-1}$ *are reduction simulations. We say that* P *and* Q *are reduction bisimilar, written* $P \sim_r Q$ *iff* $(P, Q) \in \mathcal{R}$ *for some reduction bisimulation* $\mathcal{R}$.

Definition 1.4.11 (Strong Bisimilarity)**.** *A strong simulation is a binary relation* $\mathcal{R}$ *satisfying the following:* $(P, Q) \in \mathcal{R}$ *implies that:*

- *if* $P \xrightarrow{\alpha} P'$ *then* $\exists Q' : Q \xrightarrow{\alpha} Q' \wedge (P', Q') \in \mathcal{R}$.

The relation $\mathcal{R}$ *is a strong bisimulation iff both* $\mathcal{R}$ *and its converse* $\mathcal{R}^{-1}$ *are strong simulations. We say that* P *and* Q *are strongly bisimilar, written* $P \sim Q$ *iff* $(P, Q) \in \mathcal{R}$ *for some strong bisimulation* $\mathcal{R}$.

Definition 1.4.12 (Weak Bisimilarity)**.** *A (weak) simulation is a binary relation* $\mathcal{R}$ *satisfying the following:* $(P, Q) \in \mathcal{R}$ *implies that:*

- *if* $P \stackrel{s}{\Longrightarrow} P'$ *where* $s \in \mathcal{L}^*$ *then* $\exists Q' : Q \stackrel{s}{\Longrightarrow} Q' \wedge (P', Q') \in \mathcal{R}$.

The relation $\mathcal{R}$ *is a bisimulation iff both* $\mathcal{R}$ *and its converse* $\mathcal{R}^{-1}$ *are simulations. We say that* P *and* Q *are (weakly) bisimilar, written* $P \approx Q$ *iff* $(P, Q) \in \mathcal{R}$ *for some bisimulation* $\mathcal{R}$.

Language and failures equivalences

We shall use the notion of language and failures in order to measure the expressive power of the calculi. Language notion is particularly suitable for this work, because the comparison involves different computability models.

Following [BBK93], we say that a process generates a sequence of non-silent actions s if it can perform the actions of s in a finite maximal sequence of transitions. More precisely:

Definition 1.4.13 (Sequence and language generation)**.** *The process* P *generates a sequence* $s \in \mathcal{L}^*$ *if and only if there exists* Q *such that* $P \stackrel{s}{\Longrightarrow} Q$ *and* $Q \not\xrightarrow{\alpha}$ *for any* $\alpha \in Act$. *Define the language of (or generated by) a process* P, $L(P)$, *as the set of all sequences* P *generates. We say that* P *and* Q *are language equivalent , written* $P \sim_L Q$ *, iff* $L(P) = L(Q)$.

The above definition basically states that a sequence is generated when no transition rules can be applied. It is clearly related to the notion of language generation of models of computation. Namely, formal grammars where a sequence is generated when no rewriting rules can be applied.

We recall the notion of *failure* following [Mil89]. We first need the following notion:

Definition 1.4.14. *We say that P is stable iff $P \not\xrightarrow{\tau}$.*

Intuitively we say that a pair $\langle e, L\rangle$, with $e \in \mathcal{L}^*$ and $L \subseteq \mathcal{L}$, is a failure of P if P can perform e and thereby reach a state in which no further action (including τ) is possible if the environment will only allow actions in L.

Definition 1.4.15 (Failures)**.** *A pair $\langle e, L\rangle$, where $e \in \mathcal{L}^*$ and $L \subseteq \mathcal{L}$, is a failure of P iff there is P' such that: (1) $P \stackrel{e}{\Longrightarrow} P'$, (2) $P' \not\xrightarrow{l}$ for all $l \in L$, and (3) P' is stable. Define Failures(P) as the set of failures of a process P. We say that P and Q are failures equivalent , written $P \sim_F Q$ iff Failures(P) = Failures(Q).*

State of the Art

This project aims at analyzing the reachability problem in CCS$_!$, hence it is important to highlight other similar theories developed for the most representative process calculi, like CCS and π-calculus. Moreover, it is important to know which tools using this theories have been developed. In the following section, those aspects are going to be presented.

Scientific Background We will present a short abstract of three works that are related with our project

- **On the expressivity of infinite and local behavior in fragments of the π-calculus**[Ara09]: This Ph.D. thesis is an expressiveness study for fragments of the π-calculus. The main results of this dissertation includes a characterization of CCS$_!$ in the Chomsky Hierarchy, i.e., the author propose a way to generate formal languages using CCS$_!$ processes. Besides, they prove the relationship between context-free grammars and the CCS$_!$ processes by showing that not all the context-free grammars can be generated by those processes (preserving termination). As stated before, the work proposed here is based on the results from such study, since the relation between the formal languages, the language generation and the reachability problem is direct. Then, our results will help to prove other important aspects on the expressiveness of π-calculus.

- **On the decidability of the control reachability problem in the asynchronous π-calculus**[AM02]: This work studies an analog problem, it consists basically that given a process in asynchronous π-calculus (with some special constraints) they need to determine if such process can reach a special configuration. The main difference lies in the calculus, since they use an strictly asynchronous formalism then their results do not apply directly in our problem.

- **On the expressive power of recursion, replication and iteration in process calculi**[BGZ09]: This report provides a wide analysis about three mechanisms for giving infinite behavior in process calculi, namely they study the expressiveness of recursion, iteration and replication. The authors provide an hierarchy between the three mechanisms, they give an encoding from recursion to replication, and replication to iteration. They also study four basic issues, termination (all computations of a given process are finite), convergence (the existence of a finite computation), barb (the possibility of performing an action after a sequences of synchronizations) and weak bisimulation (check whether two processes are weak bisimilar). In this work, we shall use the decidability of barb in the replication case.

Technological Background In this section, we present two practical tools related to process calculi

- **The Edinburgh Concurrency Workbench (CWB):**[3] This workbench is an automated tool which caters for the manipulation and analysis of concurrent systems. In particular, the CWB allows for various equivalence, preorder and model checking using a variety of different process semantics. For example, with the CWB it is possible to:
  - define behaviors given either in an extended version of CCS or in SCCS, and perform various analyses on these behaviors, such as analyzing the state space of a given process, or checking various semantic equivalences and preorders;
  - define propositions in a powerful modal logic and check whether a given process satisfies a specification formulated in this logic;
  - play Stirling-style model-checking games to understand why a process does or does not satisfy a formula;
  - derive automatically logical formulae which distinguish nonequivalent processes;
  - interactively simulate the behavior of an agent, thus guiding it through its state space in a controlled fashion.
- **The Mobility Workbench (MWB):**[4] MWB is a similar to the previous tool, but its application is focused on π-calculus instead of CCS. In an analogous way this workbench is used to model concurrent systems and it allows to reason about equivalences, behaviors, among other functionalities.

[3] http://homepages.inf.ed.ac.uk/perdita/cwb/summary.html
[4] http://www.it.uu.se/research/group/mobility/mwb

1.4.2 Project Context

This project is part of a much bigger project named FORCES[5]. It is a Colombian-French project funded by the program of "Equipes Associees" of INRIA[6]. The teams involved in this research collaboration are COMÈTE[7] (INRIA), the Music Representation Research Group (IRCAM)[8] and AVISPA Research Group[9] (Universidad del Valle in agreement with Universidad Javeriana at Cali, Colombia -*which the researcher is member*-) (Colciencias). The main goal is to provide more robust formalisms for analyzing the emergent systems that our teams have been modeling during recent years: I.e., Security Protocols, Biological Systems and Multimedia Semantic Interaction.

As it can be seen, the application of process calculus such as π-calculus is important in different areas. This project will help for developing and implementing new tools based in this calculus, since determining if an action can be made is central for many models in different areas. The main advantage will be given in security analysis which is a very important topic nowadays.

Then, the results will reinforce the work made in FORCES and also they will give a new result in concurrency theory. An additional advantage of this project is to consolidate relationship between AVISPA and École Polytechnique at Paris, through teacher Frank Valencia (Project Co-Supervisor), since this project feeds the interests that have been built-up jointly.

[5] *FORmalisms from Concurrency for Emergent Systems.* More info at: http://www.lix.polytechnique.fr/comete/Forces/Welcome.html

[6] *Institut national de recherche en informatique et automatique.* More info at: http://www.inria.fr/

[7] http://www.lix.polytechnique.fr/comete/

[8] *Institut de Recherche et Coordination Acoustique/Musique.* More info at: http://www.ircam.fr/

[9] http://cic.puj.edu.co/wiki/doku.php?id=grupos:avispa:avispa

1.5 Contributions

The main contributions of our work are listed below:

1. We provide an accurate analysis for the relation between the local names and the observable actions by showing an encoding that reveals the logarithmic relation between them.

2. We give an impossiblility result concerning with the use of a fixed amount of local names in a process. In consequence, we cannot impose a delay of an action progressively growing using a constant amount of nested restrictions.

3. We show a special case of the relation between nested restrictions and observable actions, where there is no process with two nested restrictions which can delay the execution of an action by three previous actions.

4. We prove the direct relation between nested local names and sequence of observable actions.

5. We present a transformation of the reachability problem to barbs. In consequence, we prove the decidability of the reachability problem for $\text{CCS}_!$.

6. We provide an algorithmic approach for solving the reachability problem and its respective computational complexity.

7. We present a small interpreter for $\text{CCS}_!$. It receives a process written in $\text{CCS}_!$ and returns its possible evolutions according to the operational semantics.

Chapter 2

Local Names vs. Observable Actions

In this chapter we describe the influence of the local names on the amount of observable actions that a process can perform before reaching a specific state. First, we shall introduce the notion of a special family of CCS$_!$ processes called trios and its relationship with the dependency of actions. We then show how to use this kind of processes to analyze the relation between local names and observable actions. Then, we prove the limit of the local names when trying to perform observable actions before reaching a special state. We after relate the decidability of barbs with reachability problem and finally we use this relation and our result to determine the decidability of the reachability problem.

2.1 A Special Family of Processes: Trios

Trios-Processes [Ara09] is a special family of CCS$_!$ processes with a simple but powerful structure which conserves the behavior of CCS$_!$ processes. Its structure will allow us to analyze more easily the dependency between actions.

2.1.1 Formal Definition

Intuitively, trios processes are a subset of arbitrary CCS$_!$ processes composed only by "trios" (sequences of three actions). Formally we have:

Definition 2.1.1 (Trios Process)**.** *We shall say that a CCS$_!$ process T is a trios-process iff all prefixes in T are trios; i.e., they all have the form $\alpha.\beta.\gamma$ and satisfy the following: If $\alpha \neq \tau$ then α is a name bound in T, and similarly if $\gamma \neq \tau$ then γ is a co-name bound in T. For instance $(\nu l)(\tau.\tau.\overline{l} \mid l.a.\tau)$ is a trios-process. We will view a trio $l.\beta.\overline{l}$ as linkable node with incoming link l from another trio, outgoing link $\overline{l}$ to another trio, and contents β.*

Interestingly, the family of trios-processes can capture the behaviour of arbitrary CCS$_!$ processes via the following encoding:

Definition 2.1.2. *Given a CCS$_!$ process P, $[\![P]\!]$ is the trios-process $(\nu\, l)(\tau.\tau.\overline{l} \mid [\![P]\!]_l)$ where $[\![P]\!]_l$, with $l \notin n(P)$, is inductively defined as follows:*

$$
\begin{array}{ll}
[\![0]\!]_l = & 0 \\
[\![\alpha.P]\!]_l = & (\nu\, l')(l.\alpha.\overline{l'} \mid [\![P]\!]_{l'}) \text{ where } l' \notin n(P) \\
[\![P \mid Q]\!]_l = & (\nu\, l', l'')(l.\overline{l}.\overline{l''} \mid [\![P]\!]_{l'} \mid [\![Q]\!]_{l''}) \text{ where } l', l'' \notin n(P) \cup n(Q) \\
[\![!P]\!]_l = & (\nu\, l')(!l.\overline{l'}.\overline{l} \mid ![\![P]\!]_{l'}) \text{ where } l' \notin n(P) \\
[\![(\nu\, x)P]\!]_l = & (\nu\, x)[\![P]\!]_l
\end{array}
$$

Notice that the trios-process $[\![\alpha.P]\!]_l$ encodes a process $\alpha.P$ much like a linked list. Intuitively, the trio $l.\alpha.\overline{l'}$ has an outgoing link l to its continuation $[\![P]\!]_{l'}$ and incoming link l from some previous trio. The other cases can be explained analogously. Clearly the encoding introduces additional actions but they are all silent—i.e., they are synchronisations on the bound names l, l' and l''.

Unfortunately the above encoding is not invariant w.r.t. language equivalence because the replicated trio in $[\![!P]\!]_l$ introduces divergence. E.g, $L((\nu x)!x) = \{\epsilon\}$ but $L([\![(\nu x)!x]\!]) = \emptyset$. It has, however, a pleasant invariant property: weak bisimilarity, $\approx$.

Proposition 2.1.1. *For every CCS$_!$ process P, $P \approx [\![P]\!]$ where $[\![P]\!]$ is the trios-process constructed from P as in Definition 2.1.2.*

2.1.2 Action Dependency in Trios

Intuitively, the reachability problem consists in checking whether a process can perform a specific action at some point in its evolution. Then we are specially interested in the sequences that a process can perform before such action, since we shall use the relationship between these sequences and the structure of the process in order to obtain an useful measure for determining reachability. Now we shall present the influence of the dependencies in a trios process.

For analyzing the influence of dependencies, let $s = a.b.c$ and $s' = a^{10}b$ be sequences of actions. Now take a trios-process P and see it as a linked list, it is easy to check that if $P \stackrel{s}{\Longrightarrow}$ then the there must be a "link" between a and b, and similarly between b and c, since the observable actions can only be in the center of a trio (as data in a linked list). On the other hand, take a process Q s.t. $Q \stackrel{s'}{\Longrightarrow}$, this case is different from s', since there is no order between the a's thus what you can only infer from the structure of the process is that you need a special link after performing a^{10}, and the independence of the a's implies that they can be performed anytime.

This intuitive description gives us a powerful insight at analyzing the sequences that a trios-process can perform. From this fact we can say that the case when you need a maximum amount of "links" is when there is a complete dependency among actions. Therefore, when the links are used in the worst way, the result is a relation 1-1 between observable actions and local names. This is not the case when there exist independent actions.

In the next section we will focus mainly on the behavior of trios-processes with independent sequences in order to find the best way of exploiting local names, and then figuring out the limits of the possible sequences that a process can perform given an amount of local names.

2.2 Exploring the Limits of Local Names

In this section we shall show the limits of the local names when we need to build a process which performs a set of actions before reaching a specific state. Using trios-processes, we will be able to show the optimal way of using the local names. Later, we shall prove that using a fixed number of local names will not allow to have an unbounded amount of actions, i.e., at some point we will obtain an undesirable behavior. Finally, we shall show a specific case of this limit.

2.2.1 Logarithmic Local Names for Independent Actions

As said before, we will focus in minimizing the use of local names in order to find the relation between local names and sequence of observable actions. The intuitive idea is to build a trios-process T which performs a sequence of independent actions $s = a^n$ then $\beta = b$ using as least local names as possible.

We propose a logarithmic amount of local names with respect to n. Let $v_n \in \mathbb{N}$ the number of local names used; $t_n \in \mathbb{N}$ the number of resulting trios, $f(i,n) \in \mathbb{N}\times\mathbb{N}$ a function (used for convenience) that takes the half of t_n, i times; $\prod_i P$ denotes the parallel composition of P, i times; and form now on, we will use l_i to denote local names in the process.

$$v_n = \lceil \log_2 n \rceil + 1 \quad t_n = 2^{V_n} \quad f(i,n) = 2^{(v_n - i)}$$

Then, we construct T in four parts. In the first part (P_1), every a is encoded in a trios-process of the form $\tau.a.\bar{l_1}$. The second part (P_2) is used when n is not a power of 2, then we need to fill the output actions that left to complete an amount of $\bar{l_1}$ that must be a power of 2 (This will be used for making one reception for two variables in a progressive way). The third part (P_3) consists in the reception of every output name $\bar{l_1}$ and every trio finish with an acknowledge $\bar{l_2}$. For the first n

trios (which output in $\bar{l}_1$) we need the same amount of receptions, then this process is made successively, thus the amount of new trios is reduced to the half until we get just one trio which outputs in the channel ($\bar{l}_{v_n}$). The last part (P_4) consists in the last reception that represents the acknowledge of all the a's previously specified, hence the b can be executed for having $a^n b$.

$$P_1 = \prod_n \tau.a.\bar{l}_1 \tag{2.1}$$

$$P_2 = \prod_{f(1,n)-n} \tau.\tau.\bar{l}_1 \tag{2.2}$$

$$P_3 = \prod_{i=2}^{v_n} \prod_{f(i,n)} l_{i-1}.l_{i-1}.\bar{l}_i \tag{2.3}$$

$$P_4 = l_{v_n}.b.\tau \tag{2.4}$$

$$T = P_1 \mid P_2 \mid P_3 \mid P_4$$

For illustrating the process proposed above, let us show the process that results for $a^4 b$, $n = 4$.

$$v_4 = \lceil \log_2 4 \rceil + 1 = 3 \quad t_4 = 2^{v_4} = 8$$
$$f(i,n) = 2^{(v_n - i)} \quad f(1,4) = 4,\ f(2,4) = 2,\ f(3,4) = 1$$

$$P_1 = \prod_4 \tau.a.\bar{l}_1 = \tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1 \tag{2.5}$$

$$P_2 = \prod_{f(1,4)-4} \tau.\tau.\bar{l}_1 = \prod_0 \tau.\tau.\bar{l}_1 = 0 \tag{2.6}$$

$$P_3 = \prod_{i=2}^{3} \prod_{f(i,4)} l_{i-1}.l_{i-1}.\bar{l}_i = \prod_{f(2,4)} l_1.l_1.\bar{l}_2 \mid \prod_{f(3,4)} l_2.l_2.\bar{l}_3 \tag{2.7}$$

$$= \prod_2 l_1.l_1.\bar{l}_2 \mid \prod_1 l_2.l_2.\bar{l}_3 \tag{2.8}$$

$$= l_1.l_1.\bar{l}_2 \mid l_1.l_1.\bar{l}_2 \mid l_2.l_2.\bar{l}_3 \tag{2.9}$$

$$P_4 = l_3.b.\tau \tag{2.10}$$

$$Q = P_1 \mid P_2 \mid P_3 \mid P_4 \tag{2.11}$$

$$= (\tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1 \mid \tau.a.\bar{l}_1) \mid (l_1.l_1.\bar{l}_2 \mid l_1.l_1.\bar{l}_2 \mid l_2.l_2.\bar{l}_3) \mid l_3.b.\tau \tag{2.12}$$

This construction uses a logarithmic amount of local names with respect to the amount of observable actions. From now on, we shall show that this is the best way of using local variables in order obtain the longest sequence of independent observable actions before reaching a subprocess which is able to perform a particular action (in this case b).

2.2.2 Constant Local Names and Unbounded Independent Actions

In order to prove that the previous encoding uses the local names in an optimal way, first we will proceed by proving that using a fixed number of local names (less than logarithmic) we cannot build processes which delay an action unboundedly, i.e., there is no amount of local names s.t. we can unboundedly construct processes where the action (we are analyzing) can only be executed after a growing number of observable actions.

Moreover, we want to prove that a constant number of local names constrains the expressiveness of the calculus, we shall show that if we only use a fixed number of local variables then there is a limit for the processes that we can build. We will begin by providing some results about sets and its relation with infinite sequences. Then, we will introduce a lemma that proves the limits of using fixed local names.

Remark 2.2.1. *We are interested in infinite sequences using natural numbers as elements, then from now on the alphabet for constructing the sequences will be the natural numbers. Besides, we will use $x \in s$ to indicate that the number x is in the sequence s (as a symbol taken from the alphabet).*

This kind of sequences will be very useful for analyzing the infinite behavior when we try to generate an infinite amount of processes with fixed local names. The next result is from sets, but it will be helpful when we relate it to infinite sequences.

Theorem 2.2.1. *[Apo67]* ***(The Well-Ordering Principle)*** *Every non-empty set of positive integers contains a smallest element.*

Now we will state the relationship between a set and and an infinite sequence, basically such sequences can be seen as a set with the same elements but without repeated elements and order. The set might not be infinite, but it is irrelevant for our purposes. In consequence, we can make a relation between the result of the "well-ordering principle" in sets and infinite sequences of naturals as a corollary.

Corollary 1. *Every non-empty infinite sequence of naturals contains a smallest element.*

We have related the results from sets to infinite sequences of naturals, now we introduce a convention for the smallest value in a sequence and the definition of "delay" an specific action.

Definition 2.2.1. *Let s be a non-empty infinite sequence, we use $min(s)$ to denote a function that takes an infinite of naturals and gives the smallest value in s and it exists by 1.*

Definition 2.2.2 (Delay)**.** *A process P delays β with a minimum number of k actions defined over a finite alphabet Σ iff $P \stackrel{s.\beta}{\Longrightarrow}$ where $s = a_1 \dots a_k$ and there is no k' where $k' < k$ and $P \stackrel{s'.\beta}{\Longrightarrow}$ with $s' = b_1 \dots b_{k'}$, and $s, s' \in \Sigma^*$. From now on, $P_{k,\beta,\Sigma}$ denotes a process P which produces β with a minimum number of k actions defined over a finite alphabet Σ.*

Now we can present the general result about fixed local names, we will prove a lemma that states that if we only have a specific amount of local variables then we are not able to build every process $P_{k,\beta,\Sigma}$ with $k \geq 0$.

This impossibility is based in the idea that if there are finite local variables, then all processes can only be constructed using a finite amount of possible trios, as consequence every process can be seen as a tuple since we can count the ocurrences of the trios and save such values in the tuple.

The result is an infinite sequence of tuples and we can order it to find two subsequent processes (say $P_{i,\beta,\Sigma}$ and $P_{j,\beta,\Sigma}$) which are supposed to have a minimum delay for the β action (i and j actions respectively), but this order means that the "greater" process $P_{j,\beta,\Sigma}$ can do the same things as the "smaller" $P_{i,\beta,\Sigma}$ then we can conclude that $P_{j,\beta,\Sigma}$ could finish before than planned (after i actions) leaving us in a contradiction. After introducing intuition, the lemma is formally stated.

Lemma 1. *Let S be defined as $\{P_{k,\beta,\Sigma} \in Trios \mid k \geq 0\}$, there is no n_ν s.t for every $P \in S$ then $|bn(P)| \leq n_\nu$.*

Proof. Suppose by means of contradiction that there exists n_ν such that for every $P \in S$ then $|bn(P)| \leq n_\nu$. The number of possible trios N is finite since actions (alphabet, local names and τ) are finite. Hence, we have the finite set $L_T = \{t_1, t_2, ..., t_j, ..., t_N\}$ where every t_j is a trio and L_T represent all possible trios.

If the trios are finite then every $P_{k,\beta,\Sigma}$ can be written in a normal form $P_{k,\beta,\Sigma} = (\nu \overrightarrow{x}) \prod_i \alpha_i.\beta_i.\gamma_i$ where every $\alpha_i.\beta_i.\gamma_i \in L_T$. It is important to notice that $P_{k,\beta,\Sigma}$ represents the family of processes with a delay of k actions for β, but we are only interested in the minimum process of such family (ommiting unnecesary trios).

Every $P_{k,\beta,\Sigma}$ can be seen as a tuple $p_k = (t_1^k, t_2^k, ..., t_j^k, ..., t_N^k)$ where t_j^k represents the amount of t_j present in $P_{k,\beta,\Sigma}$. Hence, S can be represented as an infinite sequence $I = p_1.p_2 \dots p_k \dots$, the idea is to find a subsequence of I (say I') that is nondecreasing ordered with respect to every field (t_i where $1 \leq i \leq N$). This can be done by taking the projection of every field as an infinite sequence and filter it to obtain a nondecreasing order.

Let $L = T_1.T_2 \dots$ be an infinite sequence of tuples, $L^i = T_1^i.T_2^i \dots$ be the projection of the field i in every tuple of the sequence and N the number of fields (as much as possible trios). Then we have the family of functions $f^i : (\pi)^\omega \to (\pi)^\omega$ and $F : (\pi)^\omega \to (\pi)^\omega$, these functions must satisfy the following equations:

$$F(L) = f^N \circ \dots \circ f^1(L)$$
$$f^i(L) = T_j.f^i(L')$$
$$\text{where } T_j^i = min(L^i) \text{ and } L = T_1 \dots T_j.L'$$

Therefore, $I'_{ord} = F(I)$ is an infinite and nondecreasing ordered subsequence of I. From I'_{ord}, we can take two subsequent elements say $p_i = (t_1^i, t_2^i, ..., t_N^i)$ and $p_j = (t_1^j, t_2^j, ..., t_N^j)$ where for every $t_m^i \leq t_m^j$ ($1 \leq m \leq \mathbb{N}$) and $i < j$.

By hypothesis there exist a sequence $s' = \alpha_1 \dots \alpha_i$ such that $P_{i,\beta,\Sigma} \overset{s'.\beta}{\Longrightarrow}$, and for every sequence s if $P_{i,\beta,\Sigma} \overset{s}{\Longrightarrow}$ then $P_{j,\beta,\Sigma} \overset{s}{\Longrightarrow}$ since for every trio $t = \alpha.\beta.\gamma$ if $t \in P_{i,\beta,\Sigma}$ then $t \in P_{j,\beta,\Sigma}$. Hence, $P_{j,\beta,\Sigma} \overset{s'.\beta}{\Longrightarrow}$ but $P_{j,\beta,\Sigma}$ was supposed to delay β with j actions minimum, leaving us in a contradiction. As a result, the set S cannot be obtained with a fixed number of local variables. □

In conclusion, we cannot get the set S since in some point a process will have a lesser delay than supposed. This result shows that we can give an specific behavior to a process depending in the amount of variables that we have, since in some point we will be limited to express a different behavior if we have already used all the process that are available. Therefore, if we can control the number of local variables then we can have more expressiveness since could have more processes that can be built.

2.2.3 Two Local Names for Three Independent Actions

The previous result proved the limit of fixed local names, now we present a particular case where we try to build a process which is able to delay b with a^3 minimum. This is a proof by cases where we show that there is no possible combination of trios in order to obtained the desired behavior.

α	Case	β	Case
τ	Ok	τ	Could not be executed
l_1	1	$\bar{l}_1$	Ok
l_2	2	$\bar{l}_2$	Generates ab

Figure 2.1: Three trios of the form $\alpha.a.\beta$

α	Case
τ	Generates b
l_1	Generates ab
l_2	3

Figure 2.2: One trio containing $\bar{l}_2$ (Case A, $\alpha.\bar{l}_2.\beta$)

Theorem 2.2.2. *There is no CCS$_!$ trios-process P with two nested local variables l_1 and l_2 that delays the execution of an action b with a^3 independent actions at least.*

Proof. We want to prove that there is no CCS$_!$ trios-process which execute the b action after 3 independent a's. We will proceed by contradiction, then suppose that exists a trios-process T that delays b with a^3.

Suppose that we can build a trios-process T that can only output the sequence a^3b where b depends from the three a's and they are independent among themselves. Consider two nested local names (l_1, l_2), then T has to be composed by the following trios:

1. There must be a final trio of the form $l_2.b.\tau$, where l_2 will be synchronized with an $\bar{l}_2$ action when the three a's have been executed.

2. On the other side, there must be three trios of the form $\alpha.a.\beta$. Where $\alpha \in \{\tau, l_1, l_2\}$ and $\beta \in \{\tau, \bar{l}_1, \bar{l}_2\}$, thus let us consider the values of α and β by cases. See figures 2.1, 2.2 and 2.3.

α	β	Case	α	β	Case	α	β	Case
τ	τ	Generates b	l_1	τ	Generates ab	l_2	τ	3
	l_1	Generates ab		l_1	Generates a^2b		l_1	3
	l_2	4		l_2	4		l_2	3
	$\bar{l}_1$	5		$\bar{l}_1$	5		$\bar{l}_1$	3
	$\bar{l}_2$	Generates b		$\bar{l}_2$	Generates b		$\bar{l}_2$	3

Figure 2.3: One trio containing $\bar{l}_2$ (Case B, $\alpha.\beta.\bar{l}_2$)

Case 1 ($t_1 = l_1.a.\beta$) By definition the three a's are independent, then we do not need to prefix any a thus we can replace l_1 by τ.

Case 2 ($t_2 = l_2.a.\beta$) We will prove that none initial trio can have the form $l_2.a.\beta$. By definition there exist a trio $t_f = l_2.b.\tau$ and there must be a $\bar{l}_2$ that allows the execution of t_2. In the moment that we can execute t_2 we can also execute t_f and ommit one a. Then b can be executed before than planned, a contradiction.

Case 3 ($t_3 = l_2.\alpha.\beta$) We will prove that none intermediate trio can have the form $l_2.\alpha.\beta$. Suppose by means of contradiction that such trio exists and will be synchronized with $t_f = l_2.b.\tau$. By definition $\alpha = \bar{l}_2$ or $\beta = \bar{l}_2$ and by hypothesis t_3 is the "last" trio before t_f and after the a^3. Now t_3 contains a l_2 then there must be another trio containing a $\bar{l}_2$ for synchronizing with this action, but this action can also be synchronized with t_f and it would be the "last" trio before t_f, leaving us in a contradiction.

Case 4 ($t_4 = \alpha.l_2.\bar{l}_2$) We will prove that none intermediate trio can have the form $\alpha.l_2.\bar{l}_2$. By definition there exist al least one trio $\tau.a.\bar{l}_1$ and $\alpha \in \{\tau, l_1, l_2\}$. If $\alpha = l_2$ we are in case 2, else if $\alpha = \tau$ we can execute it then we will obtain $l_2.\bar{l}_2$ analog to case 2 and if $\alpha = l_1$ we can execute $\tau.a.\bar{l}_1$ and the result is also $l_2.\bar{l}_2$ analog to case 2.

Case 5 ($t_5 = \alpha.\bar{l}_1.\bar{l}_2$) We will prove that none intermediate trio can have the form $\alpha.\bar{l}_1.\bar{l}_2$. By definition $\alpha \in \{\tau, l_1\}$ and there exist al least one trio $\tau.a.\bar{l}_1$. If $\alpha = \tau$ then we can execute it and the result is $\bar{l}_1.\bar{l}_2$, else if $\alpha = l_1$ we can execute $\tau.a.\bar{l}_1$ and the result is also $\bar{l}_1.\bar{l}_2$. Besides, there exist another trio $\tau.a.\bar{l}_1$ then there must be a trio that contains l_1 that will be synchronized with it, but this action can also be synchronized with $\bar{l}_1.\bar{l}_2$ and after that we can execute $l_2.b.\tau$. As consequence, we can execute ab or a^2b (depending of α), then b can be performed before than planned, a contradiction.

□

2.3 The Reachability Problem

2.3.1 Optimizing the Use of Local Names

In this section we shall show our main result that relates the amount of nested local variables and the amount of observable actions that can be performed before reaching a state. This relation will be based on the representation of a trios process as a binary tree.

First we introduce the notion of dependency and its relation with trios process by a convenient structure that we call *trio dependency tree*. This tree is basically a binary tree adapted to represent trios processes, they have some useful properties that will allow us to show the limits of the local variables. Finally we prove the relation between the amount of local variables and the observable actions by showing that the best tree that can be built has a logaritmic height with respect to the number of independent actions you want to perform.

Let us define the notion of dependency for actions and trios, and a notion of multi-set for trios processes. Intuitively, an action β depends on another action α (in a process P) if P could not had performed β without performing α first.

Definition 2.3.1 (Causal Dependency)**.** *Let P be a $CCS_!$ process, s be a sequence of actions, α and β be actions. We say that β depends on α if there exists s s.t. $P \stackrel{s.\beta}{\Longrightarrow}$ and $\alpha \in s$ (and directly depends if $s = \alpha$). If an action β does not depend on any action we say that is independent, this means that β can be performed by P anytime.*

Remark 2.3.1 (Multi-Set of Trios)**.** *We will use S_P for representing the multi-set of trios built from a trios process P. The idea is that any trios process P can be seen as a parallel composition of trios, then we can take P as a multi-set of trios. More formally, any trios-process P can be transformed into $P' = (\nu \overrightarrow{x})\ \alpha_1.\beta_1.\gamma_1 \mid \ldots \mid \alpha_m.\beta_m.\gamma_m$ and from P' we can build $S_P = \{\alpha_1.\beta_1.\gamma_1, \ldots, \alpha_m.\beta_m.\gamma_m\}$ a multi-set of trios. Notice that if P contains a replicated trio $!t$ then we can take its expansion in order to perform β (the action we want to analyze). This multi-set can be built if we only consider processes without restriction under the scope of replication $!(\nu x)$, then from now on we restrict our calculus to $CCS_!^{\neg !\nu}$.*

We are interested in finding the possible dependencies of an specific action β, such action must be in a trio of the form $\alpha.\beta.\gamma$ hence we focus on analyzing the dependencies from α. For this analysis we introduce the trio dependency trees, the idea is to take a trios process and see it as a binary tree induced by the dependencies present on the process related to α. We define three general rules for building a tree given a trios process.

Every rule takes a multi-set of trios S_P and a tree T then check if there exists a trio $t = \alpha_1.\alpha_2.\alpha_3 \in S_P$ such that some co-action of α_1, α_2, or α_3 is a leaf of T. Depending on the position of α_i in t it may discard some names. Intuitively, the rules simulate a possible execution of the dependencies in the process P, then it will check if a leaf l can be synchronized and check whether that l is being guarded by something else.

Notice that there are several ways for taking the dependencies due to the order in which the rules are applied, since there are many ways of executing the actions in a trios process. These rules reflect any possible execution of a process, i.e., if you take a trios process and apply these rules then you can build any possible execution of observable actions in the process.

More precisely, each rule takes a pair (T, S_P) and evolves to $(T', S_{P'})$ where T' has the same nodes as T but there is a leaf of T that in T' has two children and a trio is removed from S_P to obtain $S_{P'}$. This new node is built from a trio $t = \alpha_1.\alpha_2.\alpha_3 \in P$ just by checking if the co-action of α_1, α_2 or α_3 is a leaf of T. In every case, we compare a leaf with the actions present in a trio $t \in S_P$ by the first, second or third action respectively. If t is used for the synchronization then we remove a copy of t from S_P and we can apply the rules again. The next definition states formally the notion of trio dependency tree.

Definition 2.3.2 (Trio-Dependency Tree)**.** *Let P be a trios process, then a trio-dependency tree $T_{(P,t)}$ is a binary tree which represents the dependencies of a trio $t = \alpha.\beta.\gamma$ in a multi-set of trios S_P. Let $T_{(t,t)}$ be a tree with one node labeled $\bar{\gamma}$ and children labeled α and β, then $(T_{(t,t)}, S_P) \rightarrow^* (T_{(P,t)}, S_{P'}) \not\rightarrow$ where $\rightarrow^*$ is the closure of $\rightarrow$ given by the following rules:*

$$\textbf{Rule}_1 \quad \frac{T_{(Q,t)} = \overset{\gamma}{\triangle} \quad \textit{and } \alpha.\beta.\bar{\gamma} \in S_R}{(T_{(Q,t)}, S_R) \rightarrow (\overset{\gamma}{\underset{\alpha \quad \beta}{\triangle}}, S_R - \{\alpha.\beta.\bar{\gamma}\})}$$

$$\textbf{Rule}_2 \quad \frac{T_{(Q,t)} = \overset{\beta}{\triangle} \quad \textit{and } \alpha.\bar{\beta}.\gamma \in S_R}{(T_{(Q,t)}, S_R) \rightarrow (\overset{\beta}{\underset{\tau \quad \alpha}{\triangle}}, S_R - \{\alpha.\bar{\beta}.\gamma\})}$$

$$\textbf{Rule}_3 \quad \frac{T_{(Q,t)} = \overset{\alpha}{\triangle} \quad \textit{and } \bar{\alpha}.\beta.\gamma \in S_R}{(T_{(Q,t)}, S_R) \rightarrow (\overset{\alpha}{\underset{\tau \quad \tau}{\triangle}}, S_R - \{\bar{\alpha}.\beta.\gamma\})}$$

From $T_{(P,t)}$ then $L(T_{(P,t)})$ are the leaves of $T_{(P,t)}$. If l is the root of a tree $T_{(P,t)}$ then $P \overset{s}{\Longrightarrow} \overset{\bar{l}}{\longrightarrow}$ where s is a permutation of $L(T_{(P,t)})$ (leaves of $T_{(P,t)}$).

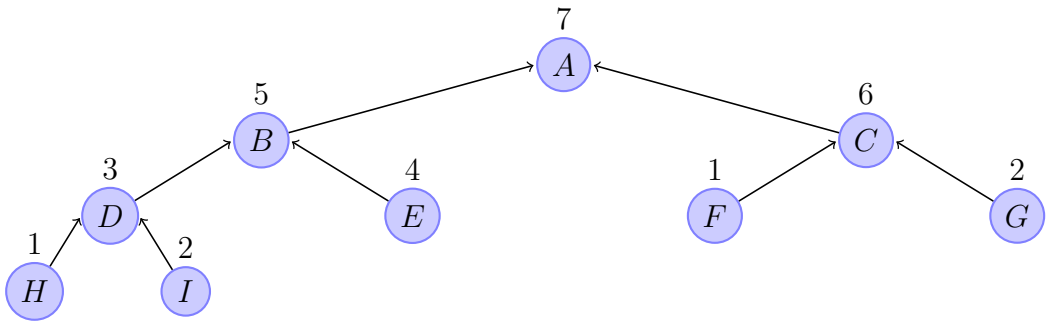


Figure 2.4: Order of execution or a trio dependency tree

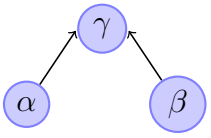


Figure 2.5: Representation of the tree in trios, subtree representing $\alpha.\beta.\bar{\gamma}$

These rules can be applied non-deterministically then from a multi-set of trios S_P and a trio $t \in S_P$ we can build different trees depending on the order that the rules are applied. The final tree represents the actions that must be performed for reaching t in S_P. First, the leaves are the actions that must be done and the nodes represent synchronizations that must be performed.

This combination will allow P to reach t, the order of execution is divided in two groups leaves and nodes (see figure 2.4). The left leaves l can be performed anytime as observable actions (unless it is τ) and right leaves must wait for its brother to be performed. The left nodes needs to be preceeded by its children to be synchronized and right nodes must also wait for its brother. Therefore, every node and leaf must be consumed to reach the root of a given trio-dependency tree T.

Moreover, if we want to read a trio-dependency tree T as a trios process P then we just take every node γ with children α and β and put a trio $\alpha.\beta.\bar{\gamma}$ in P (see figure 2.5). It is important to notice that if we have a leaf labeled with a local name at the end of the computation, then when this leaf is reached then the process will be blocked.

With these definitions we can introduce the result about the expresiveness of the local variables in a $\text{CCS}_!$ process. Intuitively, there exists a direct relation between the number of local names and the amount observable actions that can be done before releasing an specific action (say β). This relation is based in the representation of the trios in a trio dependency tree and we prove that the best

way for building this kind of tree with the least amount of variables is by using a logaritmic amount of nested local names with respect to the observable actions. We state the lemma formally.

Lemma 2. *Let Σ be a finite alphabet, $k \in \mathbb{N}$, $\beta \in \Sigma$ and $P \in CCS_!^{-!\nu}$. The minimum $|P|_\nu$ that P needs to delay β with k independent actions (definition 2.2.2) is $\lceil \log_2 k \rceil + 1$.*

Proof. Without loss of generality we will use $\beta = b$ and the sequence of k independent actions a^k where every a is independent. We proceed by stating the minimum trios that P must contain such that P delays b with k actions.

The final action must be b, then we need at least one trio $t_f = l_1.b.\tau$. Here l_1 guards the execution of b before a^k and the continuation can be τ since it does not affect the result.

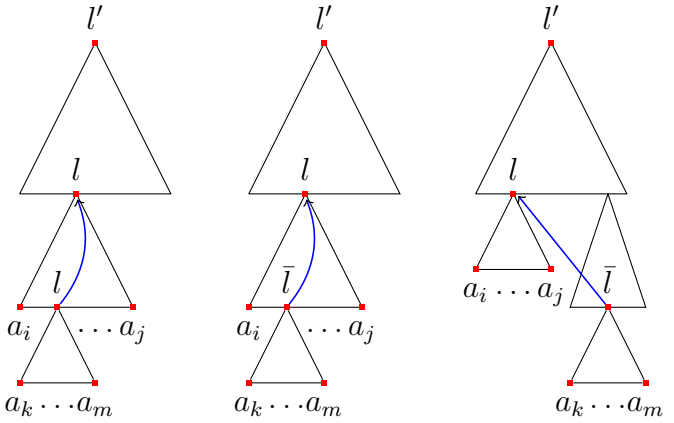
There must be at least k trios containing a, then we have $t_i = \alpha_i.a.\gamma_i$ with $0 \leq i \leq k$. Every α can be τ since the actions are independent then they can be performed anytime. Every γ must be a co-name we need to make a synchronization for ensuring the execution of every a. With the above conditions, we can state that every minimal (using the least possible local names) process Q that that delays b with a^k actions must contain the trios t_i and t_f.

Since b is the last action, then the synchronizing action $\bar{l_1}$ should be available after the synchronization of γ actions. Therefore, $\bar{l_1}$ must be guarded by other actions that allows this behavior. Then P must have (at least) a structure such that the name that guards b (l_1 in this case) is the last synchronization (if there are τ actions after b we can ommit them).

We can build a trio dependency tree where we need that l_1 be the root (the other trios must be guarding it) and we need at least k leaves contianing a's. Remember that observable actions like a must be right leaves since they cannot be in the first position of a trio. Then we need k nodes, every one with left child τ and right child a, hence the minimum tree must have at least $\lceil \log_2 k \rceil + 1$ height for containing such quantity of leaves.

Moreover, if we take any node of this tree l then none descendant of it can be l nor $\bar{l}$, since they could reach the root of the tree ommiting some leaves and reducing the delay expected. There are three cases illustrated in the figure 2.6. All of them can ommit leaves in order to reach the root.

Now suppose by means of contradiction that the minimum tree $T_{(Q,t_f)}$ contains repeated descendants (name or co-name) and the process Q associated to the tree



(a) Repeated name in descendants

(b) Repeated co-name in descendants

(c) Repeated co-name in a lower level

Figure 2.6: Tree representation for the properties

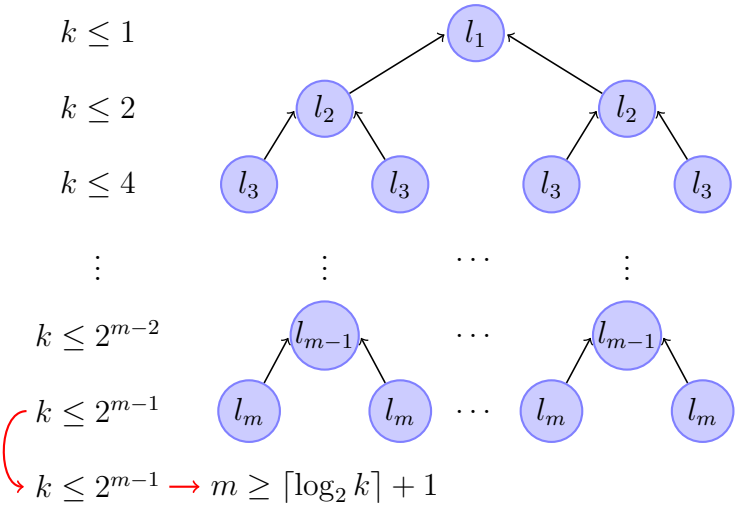


Figure 2.7: General minimum tree for $a^k b$

has delays b with miminum a^k actions. If T_Q has one repeated name in its descendants then $Q \stackrel{a^i b}{\Longrightarrow}$ where $i < k$, a contradiction.

Therefore if we want a tree T_P such that $h(T_P) = \lceil \log_2 k \rceil + 1$ and P delays b with a^k actions then every path from the root to a leaf $s = l_1.l_2 \ldots l_{k+1}$ must have a different name for every step, hence we need at least $\lceil \log_2 k \rceil + 1$ local names to obtain a process P such that P delays b with minimum a^k actions.

□

In summary, the result obtained directly relates the number of nested local names in a process and the sequences it can perform before an action. Intuitively, there is a limit on the size of the sequences related to the execution of some action in the process, and such limit says that (in th best case) we can use n local names to delay an action by 2^n independent actions at most. Using the encoding from arbitrary process to trios, the previous result also applies since it preserves weak bisimulation.

As a result, we have that given a $\text{CCS}_!^{-!\nu}$ process there is a bound on the delay we can apply to an action. Therefore, if we overpass this limit, it means that there exists another sequence that can execute the action before and this sequence is strictly related with the number of nested local names. In conclusion, this result gives us an insight on the expressiveness given by the local names and its relation with observable actions.

2.3.2 From barbs to reachability

In this section we will use the decidability result for barbs from [BGZ09] and the relation between local names and sequences, in order to give the proof of the decidability of reachability for $\text{CCS}_!^{-!\nu}$.

We need to establish the relation between barbs and reachability. Basically, the reachability problem consists in checking whether an action can be performed by a process after a sequence of observable actions, then we can put a process in parallel that has a sequence of co-actions for every element in the sequence of observables and an "acknowledge" channel at the end.

In the resulting process, every observable action in the original can be transformed into a synchronization with the co-actions in the new one. Hence, we can check that the sequence was succesfully executed if all the co-actions were consumed and the "acknowledge" was released, and actually it is a barb for the whole process. After this evolution, the action we are analyzing must be performed after a (possible empty) synchronizations, then becoming also in a barb.

Following this idea, we can now state formally the relation between barbs and reachability.

Lemma 3. *Let P and Q be a CCS$_!$ processes, s be a sequence of actions and $n \in \mathbb{N}$. $P \stackrel{s.\beta}{\Longrightarrow}$ iff $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$ where $s = \alpha_1 \dots \alpha_n$, $Q = \bar{\alpha}_1 \dots \bar{\alpha}_n.\bar{x}$ and $x \notin fn(P)$.*

Proof. ($\Rightarrow$) If $P \stackrel{s.\beta}{\Longrightarrow}$ then $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$.

By hypothesis $P \stackrel{s.\beta}{\Longrightarrow}$ then $\exists P_1 \dots P_n$ s.t. $P \stackrel{\alpha_1}{\Longrightarrow} P_1 \dots \stackrel{\alpha_n}{\Longrightarrow} P_n \stackrel{\beta}{\Longrightarrow}$ there-

fore $P \Downarrow_{\alpha_1}, \ldots, P_{n-1} \Downarrow_{\alpha_n}$ and $P_n \Downarrow_{\beta}$. On the other hand, let us consider $Q = \bar{\alpha}_1 \ldots \bar{\alpha}_n.\bar{x}$ then by definition $Q \xrightarrow{\bar{\alpha}_1} Q_1 \xrightarrow{\bar{\alpha}_2} \ldots \xrightarrow{\bar{\alpha}_n} Q_n \xrightarrow{\bar{x}}$ therefore $Q \Downarrow_{\bar{\alpha}_1}, \ldots, Q_n \Downarrow_{\bar{x}}$.Finally if we take the process $(P \mid Q)$ then $(P \mid Q) \rightarrow^* (P_1 \mid Q_1) \ldots \rightarrow^* (P_n \mid Q_n) \Downarrow_{\bar{x}} \Downarrow_{\beta}$, we can conclude that $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$.

($\Leftarrow$) If $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$ then $P \overset{s.\beta}{\Longrightarrow}$.
By hypothesis $(P \mid Q) \rightarrow^* (P_n \mid Q_n) \xrightarrow{\bar{x}} \Downarrow_{\beta}$. From the initial assumptions we can say that since $\beta \notin \mathrm{n}(Q_n)$ then $P_n \overset{\beta}{\Longrightarrow}$ and $x \notin \mathrm{fn}(P_n)$ then $Q_n \overset{\bar{x}}{\Longrightarrow}$. Now using the definition of Q we have that $Q \xrightarrow{\bar{\alpha}_1} Q_1 \xrightarrow{\bar{\alpha}_2} \ldots \xrightarrow{\bar{\alpha}_n} Q_n \xrightarrow{\bar{x}}$. In order to reach $\bar{x}$ as a barb then Q must perform $\bar{\alpha}_1 \ldots \bar{\alpha}_n$ with internal actions, then $\exists P_1 \ldots P_n$ s.t. $P \overset{\alpha_1}{\Longrightarrow} P_1 \ldots \overset{\alpha_n}{\Longrightarrow} P_n \overset{\beta}{\Longrightarrow}$. Finally we can conclude that $P \overset{s.\beta}{\Longrightarrow}$. □

The previous lemma tranforms the reachability problem into a barbs problem, now we will combine our previous result with this reduction in order to obtain the decidability of the reachability problem. Intuitively, the reachability was reduced to a problem of checking whether given a sequence and an action then the process has two special barbs, with our previous results we know that this sequences are bounded on the number of nested local names. Therefore, we only need to check all the possible sequences until the size limit (finite amount of sequences) and check if the process has a barb with any of them. Formally we have:

Theorem 2.3.1 (Decidability of Reachability Problem)**.** *Let P be a $CCS^{-!\nu}_{!}$process, Σ be a finite alphabet. The problem of deciding whether $P \overset{s.\beta}{\Longrightarrow}$ where s is a sequence over Σ^* and $\beta \in \Sigma$ is decidable.*

Proof. From lemma 3 $P \overset{s.\beta}{\Longrightarrow}$ iff $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$ where $s = \alpha_1 \ldots \alpha_n$, $Q = \bar{\alpha}_1 \ldots \bar{\alpha}_n.\bar{x}$ and $x \notin \mathrm{fn}(P)$. Besides, from [BGZ09] we know that checking $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$ for any sequence s is decidable. From lemma 2 if $P \overset{s.\beta}{\Longrightarrow}$ then $|s|$ is bounded on $|P|_{\nu}$ then the possible s are limited to those with $|s| \leq 2^{|P|_{\nu}}$ giving us a finite possible sequences S. Then the reachability depends on checking for every $s \in S$ if $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_{\beta}$, with finite S and decidable barb then $P \overset{s.\beta}{\Longrightarrow}$ is decidable. □

As the main result, we have that checking whether an action can be reached by a process (with possible infinite behavior) is decidable.

Chapter 3

Computing Reachability

In this chapter we shall present an algorithmic approach for solving the reachablity problem. As seen previously, this problem is reduced to checking some special barbs in a given process then we need to introduce the basic concepts behind the decidability of barb. First, we shall present the concept of well-structured transition system (WSTS) which is central in the barb problem. Later, we shall use the relation between $\text{CCS}_!$ and WSTS from [BGZ09] in order to present the solution proposed for the reachability problem. Finally, we will provide the solution for the reachability problem and its respective complexity analysis.

3.1 Background

In this section we will give an algorithm for determining the reachablity of an action in a given $\text{CCS}_!$ process. First, we will introduce the notion of well-structured transition systems, an useful tool for analyzing infinite state systems. Then, we shall show the result from [BGZ09] for barb decidability based on well-structured transition systems. With this notions, we will propose an algorithm for solving reachability problem. Finally, we will give a brief analysis of the complexity of the algorithm. (The background presented here was taken from [BGZ09])

3.1.1 Well-Structured Transition Systems (WSTS)

The following results and definitions are from [FS01]. Recall that a quasi-order (or, equivalently, preorder) is a reflexive and transitive relation.

Definition 3.1.1. *A well-quasi-order (wqo) is a quasi-order $\leq$ over a set X such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leq x_j$.*

Note that if $\leq$ is a wqo then any infinite sequence $x_0, x_1, x_2, \ldots$ contains an infinite increasing subsequence $x_{i_0}, x_{i_1}, x_{i_2}, \ldots$ (with $i_0 < i_1 < i_2 < \ldots$). Thus well-

quasi-orders exclude the possibility of having infinite strictly decreasing sequences.

We also need a formal definition for (infinitely branching) transition systems. This can be given as follows. Here and in the following $\rightarrow^+$ (resp. $\rightarrow^*$) denotes the transitive (resp. the reflexive and transitive) closure of the relation $\rightarrow$.

Definition 3.1.2 (Transition System)**.** *A transition system is a structure $TS = (S, \rightarrow)$, where S is a set of states and $\rightarrow \subseteq S \times S$ is a set of transitions. We define $Succ(s)$ as the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of S. We say that TS is finitely branching if, for each $s \in S$, $Succ(s)$ is finite. We also define $Pred(s)$ as the set $\{s' \in S \mid s' \rightarrow s\}$ of immediate predecessors of s, while $Pred^*(s)$ denotes the set $\{s' \in S \mid s' \rightarrow^* s\}$ (of predecessors of s).*

The functions $Succ$, $Pred$ and $Pred^*$ will be used also on sets by assuming that in this case they are defined by the point-wise extension of the above definitions. The key tool to decide several properties of computations is the notion of well-structured transition system. This is a transition systems equipped with a well-quasi-order on states which is (upward) compatible with the transition relation. Here we will use a strong version of compatibility, hence the following definition.

Definition 3.1.3 (Well-structured transition system with strong compatibility)**.** *A well-structured transition system (WSTS) with strong compatibility is a transition system $TS = (S, \rightarrow)$, equipped with a quasi-order $\leq$ on S, such that the two following conditions hold:*

1. *$\leq$ is a well-quasi order;*
2. *$\leq$ is strongly (upward) compatible with $\rightarrow$, that is, for all $s_1 \leq t_1$ and all transitions $s_1 \rightarrow s_2$, there exists a state t_2 such that $t_1 \rightarrow t_2$ and $s_2 \leq t_2$.*

3.1.2 WSTS and CCS$_!$

In [BGZ09] they prove that CCS$_!$ is a well-structured transition system. We will use their notions for solving the reachability problem, thus we shall introduce most of the concepts behind. First, they define a well-quasi-order over CCS$_!$ processes and this definition uses the following congruence:

Definition 3.1.4. *We define $\equiv_w$ as the least congruence relation satisfying the following axioms:*

$$P \mid Q \equiv_w Q \mid P$$
$$P \mid (Q \mid R) \equiv_w (P \mid Q) \mid R$$
$$P \mid 0 \equiv_w P$$

Then, they define a well-quasi-order over CCS$_!$ processes. Intuitively $P \preceq Q$ holds if Q can be obtained, up to $\equiv_w$, from P by adding some parallel processes while preserving the nesting structure given by restrictions.

Definition 3.1.5. *Let $P, Q \in CCS_!$. We write $P \preceq Q$ iff there exist $n, x_1, \ldots, x_n$, P', R, $P_1, \ldots, P_n$, $Q_1, \ldots, Q_n$ such that $P \equiv_w P' \mid \prod_{i=1}^{n}(\nu x_i)P_i$, $Q \equiv_w P' \mid R \mid \prod_{i=1}^{n}(\nu x_i)Q_i$, and $P_i \preceq Q_i$ for $i = 1, \ldots, n$.*

Now we will present some definitions from [BGZ09] relating well-structured transition systems and CCS$_!$. First, they need a notation for indicating all the sequential and bang subprocesses of a given process. They also define a special set of subprocesses using this notation.

Definition 3.1.6. *Let $P \in CCS_!$. The set $Sub(P)$ containing all the sequential and bang subprocesses of P is defined inductively as follows:*

$$\begin{aligned} Sub(\alpha.P) &= \{\alpha.P\} \cup Sub(P) \\ Sub(P + Q) &= \{P + Q\} \cup Sub(P) \cup Sub(Q) \\ Sub(P \mid Q) &= Sub(P) \cup Sub(Q) \\ Sub((\nu x)P) &= Sub(P) \\ Sub(!P) &= \{!P\} \cup Sub(P) \end{aligned}$$

With $\mathcal{P}_{P,n}$ they denote the set of all those CCS$_!$ processes whose nesting level of restrictions is not greater than n and such that their sequential subprocesses, bang subprocesses and bound names are contained in the corresponding elements of P. More precisely they propose the following definition.

Definition 3.1.7 ($\mathcal{P}_{P,n}$)**.** *Let n be a natural number and P a process. We define $\mathcal{P}_{P,n}$ as follows:*

$$\mathcal{P}_{P,n} = \{Q \in CCS_! \mid Sub(Q) \subseteq Sub(P) \wedge bn(Q) \subseteq bn(P) \wedge |Q|_\nu \leq n\}$$

The above definition will be used in the solution of the reachability problem. The following proposition is important for relating the definition of WSTS and the barb decidability.

Proposition 3.1.1. *Let $P \in CCS_!$ and $Deriv(P) = \{Q \mid P \rightarrow^* Q\}$. Then $Deriv(P) \subseteq \mathcal{P}_{P,|P|_\nu}$*

Using these definitions, they prove that $\preceq$ is a well-quasi-order.

Theorem 3.1.1. *Let $P \in CCS_!$ and $n \geq 0$. The relation $\preceq$ is a well-quasi-order over $\mathcal{P}_{P,n}$.*

In order to prove that CCS$_!$ is a WSTS, they also prove that $\preceq$ is compatible with $\rightarrow$.

Theorem 3.1.2. *Let $P, Q, P' \in CCS_!$. If $P \xrightarrow{\alpha} P'$ and $P \preceq Q$ then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \preceq Q'$.*

With the previous definitions, they define a WSTS with the reduction relation and the well-quasi-order above defined.

Theorem 3.1.3. *Let $P \in CCS_!$. Then the transition system $(Deriv(P), \rightarrow, \preceq)$ is a finitely branching well-structured transition system with strong compatibility, decidable $\preceq$ and computable Succ.*

3.1.3 Decidability of barb

In [BGZ09] they prove the decidability of the barb in $CCS_!$. Given a process, determine if it can perform, possibly after some internal moves, an observable action on a given channel. This result is based on WSTS, hence we need other definitions from this theory [FS01].

Recall that given a quasi-order $\leq$ over X, an *upward-closed* set is a subset $I \subseteq X$ such that the following holds: $\forall x, y \in X : (x \in I \wedge x \leq y) \Rightarrow y \in I$. Given $x \in X$, we define its upward closure as $\uparrow x = \{y \in X \mid x \leq y\}$. This notion can be extended to sets in the obvious way: given a set $Y \subseteq X$ we define its upward closure as $\uparrow Y = \bigcup_{y \in Y} \uparrow y$.

Definition 3.1.8 (Finite basis)**.** *A finite basis of an upward-closed set I is a finite set B such that $I = \bigcup_{x \in B} \uparrow x$.*

In this case the notion of basis is particularly important when considering the basis of the predecessor of a state in a transition system. The main interest is on effective pred-basis as defined below.

Definition 3.1.9 (Effective Pred-basis)**.** *A well-structured transition system has effective pred-basis if there exists an algorithm such that, for any any state $s \in S$, it returns the set $pb(s)$ which is a finite basis of $\uparrow Pred(\uparrow s)$.*

The following proposition is a special case of Proposition 3.5 in [FS01].

Proposition 3.1.2. *Let $TS = (S, \rightarrow, \leq)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leq$ and effective pred-basis. It is possible to compute a finite basis of $Pred^*(I)$ for any upward-closed set I given via a finite basis.*

In [BGZ09] they use this proposition for proving the decidability of barbs in $CCS_!$. They are based on the idea the next idea:

Clearly a process can perform an action α (in any number of steps) if such process is a predecessor of a process which can perform α immediately (i.e. in one step). If one can show that the set S consisting of those processes which can immediately perform α is upward closed, previous result allows us to compute effectively a finite pred-basis of $Pred^*(S)$ and therefore to decide whether a process Q belongs to $Pred^*(S)$: To this aim, in fact, it is suficient to decide whether in the (finite) basis there exist a process which is smaller than Q (this is possible because the quasi-order $\leq$ is decidable). [BGZ09]

From now on, we will use the WSTS for $\text{CCS}_!$ defined previously (but using $\mathcal{P}_{P,|P|_\nu}$ instead of $Deriv(P)$, since $Deriv(P) \subseteq \mathcal{P}_{P,|P|_\nu}$). Now we shall present the definition of the effective pred-basis for $(\mathcal{P}_{P,|P|_\nu}, \rightarrow, \preceq)$ from [BGZ09].

They start by defining the set of processes $Now_\alpha(P)$ that can immediately perform the labeled action α and which are constructed by using sequential and bang subprocesses of P (in the sense made precise by Definition 3.1.7).

Definition 3.1.10. *Let $P \in CCS_!$. The set of processes $Now_\alpha(P)$ is defined as $\{Q \in \mathcal{P}_{P,|P|_\nu} \mid Q \xrightarrow{\alpha}\}$.*

Next they show that this set is upward-closed.

Proposition 3.1.3. *Let $P \in CCS_!$. Then $Now_\alpha(P) =\uparrow Now_\alpha(P)$ holds.*

They provide a finite basis for the previously defined set. It is based on the fact that the set of sequential and bang subprocesses of a process is finite.

Definition 3.1.11. *Let $P \in CCS_!$. The set $fbNow_\alpha(P)$ is defined as follows:*

$$fbNow_\alpha(P) = \{(\nu x_1 \dots x_m)Q \mid Q \in Sub(P), m \leq |P|_\nu, \\ x_1 \dots x_m \subseteq bn(P), Q \xrightarrow{\alpha}, n(\alpha) \notin \{x_1, \dots, x_m\}\}$$

Proposition 3.1.4. *Let $P \in CCS_!$ and $\alpha \neq \tau$. Then the set $fbNow_\alpha(P)$ is a finite basis of $Now_\alpha(P)$*

Using the above definitions, they define the pred-basis as follows:

Definition 3.1.12. *Let $P \in CCS_!$. Given a process $Q \in \mathcal{P}_{P,|P|_\nu}$, we define*

$$\begin{aligned} basic_\alpha = &\{Q|R \mid R \in fbNow_\alpha(P)\} \cup \\ &\{R \in Sub(P) \mid \exists R' : R \xrightarrow{\alpha} R' \wedge Q \preceq R'\} \cup \\ &synchbasic_\alpha(Q) \end{aligned}$$

where:

$$\begin{aligned} synchbasic_\tau(Q) = \{Q_1|Q_2 \mid Q_1 \in Sub(P) \wedge Q_2 \in fbNow_\alpha(P) \wedge \\ \exists Q_1' : Q_1 \xrightarrow{\bar{\alpha}} Q_1' \wedge Q \preceq Q_1'\} \\ synchbasic_\alpha(Q) = \emptyset \text{ if } \alpha \neq \tau \end{aligned}$$

The pred-basis $pb_\alpha(Q)$ of a process Q w.r.t α is defined by induction on the structure of the process as follows:

$$\begin{aligned} pb_\alpha(0) =& basic_\alpha(0) \\ pb_\alpha(\alpha'.Q) =& basic_\alpha(\alpha'.Q) \\ pb_\alpha(Q_1 + Q_2) =& basic_\alpha(Q_1 + Q_2) \\ pb_\alpha((\nu x)Q) =& basic_\alpha((\nu x)Q') \cup \\ & \{(\nu x)Q' \mid Q' \in pb_\alpha(Q) \wedge x \neq n(\alpha)\} \\ pb_\alpha(Q_1|Q_2) =& basic_\alpha(Q_1|Q_2) \cup \\ & \{Q_1'|Q_2 \mid Q_1' \in pb_\alpha(Q_1)\} \cup \\ & \{Q_1|Q_2' \mid Q_2' \in pb_\alpha(Q_2)\} \cup \\ & sync_\alpha(Q_1, Q_2) \\ pb_\alpha(!Q) =& basic_\alpha(!Q) \\ sync_\tau(Q_1, Q_2) =& \{Q_1'|Q_2' \mid \exists \alpha \in n(P) : Q_1' \in pb_\alpha(Q_1) \wedge Q_2' \in pb_{\bar{\alpha}}(Q_2')\} \\ sync_\alpha(Q_1, Q_2) =& \emptyset \text{ if } \alpha \neq \tau \end{aligned}$$

With this definition, they provide the following result:

Lemma 4. *Let $P \in CCS_!$ and $Q \in \mathcal{P}_{P,|P|_\nu}$. Then $pb_\tau(Q)$ is a computable finite basis of $\uparrow Pred(\uparrow Q)$.*

From this lemma it is strictly forward the following theorem

Theorem 3.1.4. *Let $P \in CCS_!$. Then the transition system $(\mathcal{P}_{P,|P|_\nu}, \rightarrow, \preceq)$ is a well-structured transition system with strong compatibility, decidable $\preceq$ and effective pred-basis.*

Finally they present the decidability of barb. This result will be very useful for solving the reachability problem.

Proposition 3.1.5. *Let $P \in CCS_!$. $P \Downarrow_x$ iff $P \in Pred^*(Now_x(P))$ or $P \in Pred^*(Now_{\bar{x}}(P))$.*

Corollary 2. *Let $P \in CCS_!$. Then $P \Downarrow_x$ is decidable.*

The last result is based on the idea that the barb can be expressed as the set of processes which can perform α immediately (say $Now_\alpha(P)$) and the set of predecessors (say $Pred^*$) of them (using reduction relation). From previous results, the set Now_α is upward closed and has a finite basis, hence it is possible to compute an finite pred-basis (a basis for the possible predecessors). Therefore, we only need to compare the original process with the elements in the basis, and check whether it is "bigger" (w.r.t $\preceq$) than one of them in order to answer if it has the barb.

3.2 Solving Reachability

We can now present the algorithm for solving the reachability problem. As stated previously, asking if an action can be reached by a process is transformed in asking if the process has two special barbs, the first one is the acknowledge of a sequence of observable actions and the other is the action itself. In the same way, if we plan to describe a procedure which solve this problem, we need to consider the solution for the barb problem. Therefore, we will base our algorithm in the previous relation between barbs and WSTS in order to solve the reachability problem.

3.2.1 A procedure for reachability problem

From previous results, the reachability problem can be transformed in a barb problem for a set of sequences (built based on the structure of the process). Then, for solving this problem it is sufficient with checking these two special barbs with every sequence in the set. The first barb consists in asking whether the initial process is a predecessor (up to reduction) of a process which can perform the acknowledge action immediately.

Applying the procedure for decidability of barb, we will get a basis for this process and instead of stopping here, we can take this set to ask for the second barb in the same way. Finally we will obtain a (finite) pred-basis (a bit more complex than just one barb) and then we can check whether our process is related (w.r.t well-quasi-order) to any of the elements of the basis. If it is related, then the action is reachable, otherwise it is not.

After introducing the idea, we define a function that checks whether a process has a barb x and then checks if the resulting process (after performing this barb) has another barb y.

Definition 3.2.1. *Let P be a CCS$_!$ process, x and y actions. We define the function*

$Barb(P, x, y)$ as follows:

$$Barb(P, x, y) = \begin{cases} \texttt{true} & \text{if } P \in Pred^*(Now_x(Pred^*(Now_y(P)))); \\ \texttt{false} & \text{otherwise .} \end{cases}$$

From 3 we know that, given a process $P \in \text{CCS}_!$, then $P \overset{s.\beta}{\Longrightarrow}$ iff $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_\beta$ where $s = \alpha_1 \dots \alpha_n$, $Q = \bar{\alpha}_1 \dots \bar{\alpha}_n.\bar{x}$ and $x \notin \text{fn}(P)$. Then we give a function for this purpose, given a process, an action, and a set of sequences, checks whether any of this resulting processes have the barbs $\bar{x}$ and β.

Definition 3.2.2. *Let $P \in CCS_!$, Σ a finite alphabet, $\alpha \in \Sigma$, and S a set of sequences over Σ. We define the function $Reach(P, \beta, S)$ as follows:*

$$Reach(P, \beta, S) = \begin{cases} Barb((P \mid \bar{\alpha}_1 \dots \bar{\alpha}_n.\bar{x}), \bar{x}, \beta) & \text{if } s = \alpha_1 \dots \alpha_n, s \in S \\ \vee\ Reach(P, \beta, S - \{s\}) & \\ & \\ \texttt{false} & \text{if } S = \emptyset. \end{cases}$$

The above function allow us to provide a decision procedure for determining the reachability of an action in a process. We apply this function to the set of possible sequences that is related with the nesting restrictions. This set contains all the possible sequences (over the finite alphabet) with size less than the limit provided by 2. The following theorem relates our previous definition with the reachability problem.

Theorem 3.2.1. *Let $P \in CCS_!^{-!\nu}$, Σ a finite alphabet, $\beta \in \Sigma$, and S a set of sequences over Σ. Then $Reach(P, \beta, S)$ returns* `true` *iff there exists $s \in S$ such that $P \overset{s.\beta}{\Longrightarrow}$, where $S = \{s \mid s \in \Sigma^* \wedge |s| \leq 2^{|P|_\nu}\}$.*

Proof. From 3 $P \overset{s.\beta}{\Longrightarrow}$ iff $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_\beta$ where $s = \alpha_1 \dots \alpha_n$, $Q = \bar{\alpha}_1 \dots \bar{\alpha}_n.\bar{x}$ and $x \notin \text{fn}(P)$. Now, from 2 we know that (in the best case) we can use n nested local names to get a delay of 2^n independent actions, then it is sufficient to check if $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_\beta$ with sequences of size $2^{|P|_\nu}$ at most. By construction, all of these sequences are considered in S.

Next, we need to check for every sequence $s \in S$ if $(P \mid Q) \Downarrow_{\bar{x}} \Downarrow_\beta$. Following the same process for checking barbs, the function $Reach(P, \beta, S)$ check (for all $s \in S$) whether $R \overset{def}{=} P \mid Q$ has the barbs $\bar{x}$ and β. This is done by the function $Barb(R, \bar{x}, \beta)$, by checking if R belongs to the set $Pred^*(Now_{\bar{x}}(Pred^*(Now_\beta(P))))$. We are able to compute this set since $Now_\beta(R)$ is upward-closed 3.1.3 and then it is possible to compute a (finite) pred-basis 3.1.2 for it and then checking if the original process is a predecessor of a process which can perform β. Using this result, we can

apply the same principle (over the finite basis) in order to compute the next barb $\bar{x}$. Finally, we only need to compare if the process R is bigger (w.r.t $\preceq$) than one of the elements of the basis. Then we answer whether R has these two barbs and such procedure is repeated for every sequence in the set giving us the final result. □

Following the same idea used in [BGZ09] for deciding a barb in a CCS$_!$ process, we gave a function for checking whether an action can be performed by a process after a sequence of observable actions.

3.2.2 Analyzing the complexity

The complexity of this procedure depends basically on the amount of nested local names in a given process P, since it determines the size and the amount of sequences that we need to test in order to give a solution. It depends also on the structure of the process, since we need to compute some special sets defined by WSTS.

Therefore, the complexity of this function is divided in two parts. The first part consists on the amount of sequences, this value basically depends on the size of the alphabet $c = |\Sigma|$ (possible actions for the sequences) and the limit l given by the nested restrictions $|P|_\nu$ which is exponential w.r.t to $|P|_\nu$, then $l = 2^{|P|_\nu}$ in the worst case (sequences of independent actions). In the case of sequences with size $2^{|P|_\nu}$ we have $|\Sigma|^{2^{|P|_\nu}}$ possibilities, since we have $|\Sigma|$ for every position in the sequence. This analysis is applied with size going from 1 to $2^{|P|_\nu}$ and the result is the sum of them. Hence, the amount of sequences $|S|$ is given by the following expression:

$$\sum_{i=1}^{2^{|P|_\nu}} |\Sigma|^i$$

The second part is based on the theory of well-structured transition systems. There are no complexity analysis (so far) for this methods, we cannot say much about them. It is important to notice, that infinite-state systems can be specified by this calculus and even that, we give a finite (huge) number of steps for solving the problem.

If we put the two parts together then we have to apply a method from WSTS for every possible sequence. We know how many possibilities we have, but we do not know exactly the complexity of the methods form WSTS. Nevertheless, this algorithm becomes too complex in real life only taking into account the amount of sequences that need to be checked.

Chapter 4

CCS$_!$ Stepper

In this chapter we will present a small interpreter for CCS$_!$. It takes a process written in CCS$_!$ as input and returns the possible evolution according to the operational semantics from 1.2 and 1.3. It has been implemented using the mozart programming system[1] employing the advantages of the records in Oz. We shall describe some implementation details about the interpreter and finally we shall show some examples. This work can be the start point for a CCS$_!$ workbench like the Edinburgh Concurrency Workbench (CWB)[2] or the Mobility Workbench (MWB)[3], where it is possible to specify systems in the calculus and verify certain properties, and then explore, from the practical point of view, the application of this formalism to real life.

4.1 Grammar and General Description

In this section we shall describe some implementation details about the CCS$_!$ stepper. First we shall describe the grammar used, we have ⟨id⟩, ⟨act⟩ and ⟨ccsProc⟩. The ⟨id⟩ is an string with an initial letter and a (possible empty) of alphanumeric values, examples ⟨id⟩: "a", "a45bjbj", "P1". The ⟨act⟩ is the action that we will use inside a process, from CCS$_!$ we have that it could be an action, co-action or internal synchronization τ. We represent the co-action with the $\sim$ symbol in Oz, and with "$tau" the internal action.

Finally we have the ⟨ccsProc⟩ which represents a CCS$_!$ process, **zero** is the null process 0; **pre("a" zero)** represents the process $a.0$; **new("x" zero)** is the process $(\nu x)0$; the ⟨ccsProc⟩ **par(0 0)** is the process $0 \mid 0$ and finally **rep(0)** represents the process !0. Formally we have the grammar of figure 4.1.

[1] More info at: http://www.mozart-oz.org/

[2] http://homepages.inf.ed.ac.uk/perdita/cwb/summary.html

[3] http://www.it.uu.se/research/group/mobility/mwb

$$
\begin{aligned}
\langle \text{id} \rangle &::= \{[a-z][A-Z]\}^{+}\{[a-z][A-Z][0-9]\}^{*} \\
\langle \text{act} \rangle &::= \langle \text{id} \rangle \\
&\quad | \sim \langle \text{id} \rangle \\
&\quad | \ \$\text{tau} \\
\langle \text{ccsProc} \rangle &::= \text{zero} \\
&\quad | \ \text{pre}(\langle \text{act} \rangle \langle \text{ccsProc} \rangle) \\
&\quad | \ \text{new}(\langle \text{act} \rangle \langle \text{ccsProc} \rangle) \\
&\quad | \ \text{par}(\langle \text{ccsProc} \rangle \langle \text{ccsProc} \rangle) \\
&\quad | \ \text{rep}(\langle \text{ccsProc} \rangle)
\end{aligned}
$$

Figure 4.1: Grammar for CCS$_!$ stepper

Using this grammar we build a small interpreter that takes a CCS$_!$ process and returns its possible evolution, and it also point out which is needed to be performed in order to reach some process. For this purpose, we use another rule called ⟨step⟩ composed by an action and a process, then represents the evolution of a process by such action.

$$\langle \text{step} \rangle ::= \ \text{step(act:}\langle \text{act} \rangle \ \text{newProc:}\langle \text{ccsProc} \rangle)$$

The interpreter will take a process and return a set of steps representing the process in which the input can be transformed using the operational semantics for CCS$_!$. Since we are specially interested in trios-proceses, we also provide a function (`ProcessToTrio`) that takes a CCS$_!$ and performs the encoding from 2.1.1. This transformation can be exploited for attacking the reachability problem, but due to the complexity it becomes useless to implement the naive algorithm. Nevertheless, this can be an initial practical approach that could be optimized for real life as future work.

4.2 Tests and Examples

For performing the tests we use the function `TestEval` that takes a process written with grammar of figure 4.1. The answer is a record called `Test`, it has three fields **p**, **realResult**, and **nicePrint** where **p** is the input process, **realResult** is a set of steps that the process can perform and **nicePrint** contains the same that **realResult** but with a legible text format.

```
%% P = a.0 | (a.0 | ~a.0)
P5 = par(pre("a" zero) par(pre("a" zero) pre("~a" zero)))
{TestEval P}
% result P:
test(p:'(a.0 | (a.0 | ~a.0))'
     nicePrint:[ step(act:a newProc:'(0 | (a.0 | ~a.0))')
                 step(act:a newProc:'(a.0 | (0 | ~a.0))')
                 step(act:'~a' newProc:'(a.0 | (a.0 | 0))')
                 step(act:'$tau' newProc:'(a.0 | (0 | 0))')
                 step(act:'$tau' newProc:'(0 | (a.0 | 0))')]
     realResult:[step(act:,,,|,,, newProc:par(,,,))
                 step(act:,,,|,,, newProc:par(,,,))
                 step(act:,,,|,,, newProc:par(,,,))
                 step(act:,,,|,,, newProc:par(,,,))
                 step(act:,,,|,,, newProc:par(,,,))]

%% Q = !a.0 | ~a.0
Q = rep(par(pre("a" zero) pre("~a" zero)))
{TestEval Q}
% result Q:
% test(p:'!((a.0 | ~a.0))'
       nicePrint:[step(act:a newProc:'((0 | ~a.0) | !((a.0 | ~a.0)))')
                  step(act:'~a' newProc:'((a.0 | 0) | !((a.0 | ~a.0)))')
                  step(act:'$tau' newProc:'((0 | 0) | !((a.0 | ~a.0)))')
                  step(act:'$tau'
                       newProc:'(((0 | ~a.0) | (a.0 | 0)) | !((a.0 | ~a.0)))')
                  step(act:'$tau'
                       newProc:'(((a.0 | 0) | (0 | ~a.0)) | !((a.0 | ~a.0)))')]
       realResult:[step(act:,,,|,,, newProc:par(,,,))
                   step(act:,,,|,,, newProc:par(,,,))
                   step(act:,,,|,,, newProc:par(,,,))
                   step(act:,,,|,,, newProc:par(,,,))
                   step(act:,,,|,,, newProc:par(,,,))]
```

Figure 4.2: Examples of the $\text{CCS}_!$ stepper

The rule ⟨testProc⟩ is as follows:

$$\langle\text{testProc}\rangle ::= \text{test(p:}\langle\text{ccsProc}\rangle\ \text{p:}\langle\text{step}\rangle^*\ \text{nicePrint:}\langle\text{step}\rangle^*\text{)}$$

We provide some examples in the figure 4.2.

Chapter 5

Concluding Remarks

5.1 Summary

In chapter 2 we provide an accurate analysis for the relation between the local names and the observable actions by showing an encoding that reveals the logarithmic relation between them. Using this fact, we give an impossiblility result concerning with the use of a fixed amount of local names in a process. Therefore, we cannot impose a delay of an action progressively growing using a constant amount of nested restrictions.

In order to give an insight for the general proof, we show an special case of the relation between nested restrictions and observable actions, where there is no process with two nested restrictions which can delay the execution of an action by three previous actions. Using this results, we prove the direct relation between nested local names and sequence of observable actions.

As one of the main results, given n nested restrictions then the maximum delay for an action that can be imposed is 2^n. Finally, we present a transformation of the reachability problem to barb problem from [BGZ09]. In conclusion, we can relate the amount of sequences with the decidability of barbs in order to prove the decidability of the reachability problem for $\text{CCS}_!$.

In chapter 3 we provide an algorithmic approach for solving the reachability problem and its respective computational complexity. The resulting algorithm is much more complex than exponential order, but if we consider that we are dealing with infinite state systems then we are giving a finite number of steps for checking a property in an infinite state space.

In chapter 4 we present a small interpreter for $\text{CCS}_!$. It receives a process written

in $CCS_!$ and returns its possible evolutions. This interpreter can be the beggining of a complex system for checking properties in infinite state systems.

5.2 Future Work

The results presented in this work may have three main extensions. First, we need to explore the relation between local names and observable actions when having the full calculus, i.e, there is name generation. We think that this should not affect the result since $!(\nu x)$ does not raise the nesting value (depth) but the local names could grow in width, thus this variant could be explored. Second, we need to look for an algorithmic approach closer to real life since the complexity of the first approach is too expensive, hence it is not practical for real systems. Finally, the $CCS_!$ stepper could be extended in order to obtain a $CCS_!$ workbench for specification and verification of concurrent systems.

Bibliography

[AM02] R. Amadio and C. Meyssonnier. On decidability of the control reachability problem in the asynchronous $\pi-$calculus. *Nordic Journal of Computing*, 9(2), 2002.

[Apo67] T. M Apostol. *One-Variable Calculus, with an Introduction to Linear Algebra*, volume 1 of *Calculus*. 2nd edition, 1967.

[Ara09] Jesus Aranda. *On the Expressivity of Infinite and Local Behaviour in Fragments of the π-calculus*. PhD thesis, Ecole Polytechnique at Paris, Universidad del Valle, 2009.

[BBK93] Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J.ACM.*, 40(3):653–682, 1993.

[BGZ09] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *to appear in Mathematical Structures in Computer Science*, 2009.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science (TCS)*, 37:77–121, 1985.

[BW90] Jos. C. M. Baeten and W. Peter Weiland. *Process Algebra*. Cambridge University Press, 1990.

[CC01] D. Cacciagrano and F. Corradini. On synchronous and asynchronous communication paradigms. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *ICTCS*, volume 2202 of *Lecture Notes in Computer Science*, pages 256–268. Springer, 2001.

[CCP06] D. Cacciagrano, F. Corradini, and C. Palamidessi. Separation of synchronous and asynchronous communication via testing. *Electr. Notes Theor. Comput. Sci.*, 154(3):95–108, 2006.

[CM03] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nord. J. Comput.*, 10(2):70–98, 2003.

[FS01] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[Gor06] D. Gorla. On the relative expressive power of asynchronous communication primitives. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2006.

[Gor07] D. Gorla. Synchrony vs asynchrony in communication primitives. *Electr. Notes Theor. Comput. Sci.*, 175(3):87–108, 2007.

[GSV04] P. Giambiagi, G. Schneider, and F. D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*. Springer, 2004.

[Hoa85] C. A. R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

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