Expressiveness in biologically inspired languages

Antonio Vitale

March 2010

Department of Computer Science

University of Bologna

Mura Anteo Zamboni 7
40127 Bologna (Italy)

Dottorato di Ricerca in Informatica
Università di Bologna e Padova
INF/01 INFORMATICA
Ciclo XXII

Expressiveness in biologically inspired languages

Antonio Vitale

March 2010

Coordinatore:	Tutore:
Simone Martini	Cosimo Laneve

Abstract

A very recent and exciting new area of research is the application of Concurrency Theory tools to formalize and analyze biological systems and one of the most promising approach comes from the process algebras (process calculi).

A process calculus is a formal language that allows to describe concurrent systems and comes with well-established techniques for quantitative and qualitative analysis. Biological systems can be regarded as concurrent systems and therefore modeled by means of process calculi.

In this thesis we focus on the process calculi approach to the modeling of biological systems and investigate, mostly from a theoretical point of view, several promising bio-inspired formalisms: Brane Calculi and $\kappa-$calculus family. We provide several expressiveness results mostly by means of comparisons between calculi.

We provide a lower bound to the computational power of the non Turing complete MDB Brane Calculi by showing an encoding of a simple P-System into MDB. We address the issue of *local implementation* within the $\kappa-$calculus family: whether n-way rewrites can be simulated by binary interactions only. A solution introducing divergence is provided and we prove a deterministic solution preserving the termination property is not possible.

We use the symmetric leader election problem to test synchronization capabilities within the $\kappa-$calculus family. Several fragments of the original $\kappa-$calculus are considered and we prove an impossibility result about encoding n-way synchronization into (n-1)-way synchronization.

A similar impossibility result is obtained in a pure computer science context. We introduce CCS^n, an extension of CCS with multiple input prefixes and show, using the dining philosophers problem, that there is no reasonable encoding of CCS^{n+1} into CCS^n.

Acknowledgements

I would like to first express my sincere gratitude to my tutor Cosimo Laneve for his support and encouragement during this path.

I specially thank Nadia Busi, who I miss, for introducing me into this beautiful experience and for her friendship.

Many thanks to Catuscia Palamidessi and Luca Cardelli for their evaluation of this dissertation.

I thank my family for their patience and continuous support during these years.

I would like to thank my friend Jorge Perez for all the good time together. I thank Cristian Versari for his technical and moral support, and also for sharing the desk with me.

I also thank my house mates Eliana, Chiara and Rita.

I am indebted to many friends I met during these years in Bologna: Anna, Cinzia, Flavio, Sylvain, Marco, Pietro and Priyank.

A special mention goes to Matteo, Marta and Nicole: they surely know why.

Contents

List of Figures

Chapter 1

Introduction

One of the most recent area of convergence between Biology and Computer Science is that of Systems Biology [30]. Systems Biology is an inter-disciplinary field of Biology studying living organisms at a system level, that is centralizing the attention on the interactions of the different parts they are composed of. Systems Biology aims to develop a better understanding of the processes involved and it requires taking a systems theoretic view of biological processes analyzing inputs and outputs and the relationships between them.

A radical shift from earlier reductionist approaches, Systems Biology aims to provide a deep conceptual basis and a methodology for reasoning about biological phenomena at a molecular and cellular level. Whereas the reductionist paradigm tries to understand a complex system by gathering all the knowledge of its basic components and extrapolating their individual behaviors, Systems Biology approach states this is not enough. In order to properly characterise a complex biological system we must also comprehend its emergent behaviors and this can be accomplished only by studying the interactions among its components.

Since its creation Systems Biology has been studying the interactions between components of biological systems and how these interactions consistently produce complex behaviors in a very systematic way thus collecting a huge amount of data.

Far from being an easy task perhaps one of the most important issues is that of keeping trace of this knowledge in such a way that we can examine it and perform analysis in order to better understand biological phenomena at a system level in a consistent way. In other words one of the necessary steps for understanding the evolution of a biological system is to get an exhaustive representation of the different components and of their capability of interaction but the huge amount of informations that Systems Biology is producing must be *stored* somehow, and more important, the data must be stored in a proper way so that later it can be retrieved, used and analyzed in an efficient and easy way.

Biologists have already developed several notations and formalisms which can be used to represent, and therefore store, these informations (Object databases, Kohn charts, ODEs, chemical reactions etc.). While some of them are actually used and useful to depict even complex interactions between components the models depicted typically are not compositional or can't be animated to simulate the underlying dynamics or both in the worst case.

For example differential equations (ODEs) allow to give a complete characterization of a biological system dynamics, but they are very hard to produce, and very hard to generalize once the initial conditions are even sligthly changed; Object databases permit a scalable representation but do not contain information about the dynamics; Kohn maps [31] give a quite formal graphical description easier to understand but they also are not scalable and do not contain information on the dynamics of the system.

This is where Computer Science is trying to help Systems Biology: by providing tools for the representation and analysis of the collected information and for the formulation of consistent models of biological phenomena. The field of Computer Science providing these tools is the *Concurrency Theory*. It studies concurrent systems that are systems composed by several processes operating in *parallel* and interacting with each other.

The initial works [54, 55] showing that is possible to successfully represent and simulate biological systems at a molecular level, and the metaphor in [56] where computing parallel units are abstractions of molecules evidenced an important conjunction point between Systems Biology and Computer Science.

After those seminal works the application of formalisms coming from the Concurrency Theory to the modeling and the analysis of biological systems has attracted the interest of the scientific community in the last few years, with the perspective of formalizing, simulating and analyzing them *in silico*.

In a big-picture perspective the potential outcomes of this interdisciplinary approach are very attracting: a complete formal specification of a biological system (for example part of or an entire cell, or a portion of a tissue) would signify not only an exact, unambiguous (textual or graphical) representation of the system itself, but also the possibility of its *in silico* simulation and analysis of the interaction with other components or systems (i.e. other kind of cells, drugs, or viruses), thus drastically improving for example the approach to drugs discovery and disease prediction.

One of the most promising approach to modelling biological systems comes from the application of process algebras. Process algebras are a class of formalisms from the Concurrency Theory that offer a compositional description technique supported by apparatus for formal reasoning, moreover if we look at the organization of biological systems from an information science point of view we can easily map biological entities as processes in a process algebra. Once we model a biological system as interacting processes within a process algebra we can reason about the behaviors and properties of the system itself using well-established technique that include qualitative and quantitative analysis, and also model checking.

The first process algebra (also process calculus) applied to the modeling of biomolecular processes was the $\pi-$Calculus [57], used to formalize a signal transduction pathway. After the first use of $\pi-$Calculus the applications of process calculi to Systems Biology attracted increasing research efforts. The direct employment of $\pi-$Calculus allowed the formalization of several biological mechanisms, its variants and extension permitted the representation or analysis of cellular processes. To obtain higher abstraction level and biological faithfulness, more complex calculi have been proposed which are based on or get inspiration from $\pi-$Calculus.

The increasing number of bio-inspired process calculi is in fact indicative of an attempt to fill the expressiveness gap between process calculi and biological processes: even if biological entities can be seen as concurrent, interacting processes, the fundamental structures and mechanisms of bio-systems are obviously not originally modeled by already existing process calculi. For the sake of simplicity, each bio-inspired calculus focuses its attention on particular biological entities or mechanisms. For example, Brane calculi [13] are formalisms completely devoted to represent membrane interaction at cellular level and sublevel: they allow to easily model events like cellular reproduction, phagocytoses, viral infections. This a strongly bio-inspired formalism and the goal is the modelling of various cell dynamics together with the behaviour of biological membranes. In Brane Calculi active entities are tightly coupled to membranes and this endows membranes with the (correct) biological role of support of biochemistry; we have that computation happens on membrane and not inside of it. Another recent approach is represented by the $\kappa-$calculus [22] that has an interesting abstraction level and models molecules behaviour at domain level (complexation, activation, interaction). It was introduced in an attempt to provide a visual and compact notation for biological signaling pathways. It idealizes molecule-molecule interactions, as a particular restricted kind of graph-rewriting.

In this continuous research, once a proposed formalism presents some interesting and fresh paradigm or useful modeling property a formal study of the formalism itself takes place and typically many refining variants arise in order to improve, and better exploit, the formalism's original idea.

This was the case for both Brane Calculi and $\kappa-$calculus, and in this thesis we try to deepen their formal study by providing various expressiveness results, reading, when possible, such results from a biological point of view.

Since the thesis focus mainly on the $\kappa-$calculus family we briefly introduce it here. The κ-family originates with the formalization of the κ-calculus. The main entity is the *molecule* which is modeled as a node with fixed numbers of sites. Complexes (i.e. composition of molecules) are connected graph built over such nodes where bonds are modeled by *names*. Two kinds of rewriting rules are used to model biological reactions: *complexations*, which create bonds between sites, and *decomplexations* which are meant to destroy them. The $\kappa-$calculus has been encoded into the π calculus[41], thus potentially allowing the simulations of κ models on both uniprocessors [53] and distributed systems [62].

What makes κ an interesting language, apart from being reaction centric, is its rule-based approach to the modeling. Indeed both the rule-based and the process-based approaches avoid the combinatorial explosion affecting, for example, differential equations, but the rule-based approach has in addition the concrete advantage of making the building and modification of models easier. This is clearly shown in [19] where it is illustrated how easy could be to perform variations of a given model and why the gain in flexibility from using a rule-based approach would be even more pronounced within large models involving many entities.

A peculiar characteristic of the original $\kappa-$calculus is that the number of nodes involved in a rewriting rule is not limited. This characteristic originated an interesting problem called *local-implementation*. Quoting from [22]

> Be that in biology, or in any other decentralized computational scenario, non-local graph-rewriting takes time and more accurately it takes consensus.

Roughly speaking the local-implementation problem consists in finding, given a high level model description an encoding so that in a purely decentralized way and with binary sinchronization as the only means of communication, the encoded model is going to preserve the behavior of the original high level model description. This is an important topic because for biological systems it seems reasonable to consider atomic only binary reactions, that is interactions between at most two entities.

Contribution and Overview

In Chapter 2 we show a translation between two well known families of bio inspired calculi: the P Systems and Brane Calculi. Membrane systems (also called P systems) and Brane calculi have been recently introduced as formal models inspired by the structure and the functioning of living cells, but having in mind different goals. The aim of Membrane systems was the formal investigation of the computational nature and power of various features of the cell, while Brane calculi aims to define a model capable of a faithful and intuitive representation of various biological processes. The common background of the two formalisms and the recent growing of interests in applying P systems in

Systems Biology have raised the natural question of bridging this two research areas. This chapter goes in this direction, as it presents a direct simulation of a variant of P systems by means of Brane calculi. In particular, we consider a Brane calculus based on three operations called Mate/Bud/Drip, and we show how to use such system to simulate Simple Symport/Antiport P systems, a variant of P systems purely based on communication of objects. As an example, a simplified sodium–potassium pump modeled in Simple SA is encoded in Mate/Bud/Drip Brane calculus. The translation is one way only as it is shown how to encode *Simple SA* into MDB but not vice versa. From a purely computational point of view we may consider Simple SA as a lower bound for the computational power of MDB. The results of this chapter have been published in [61].

In Chapter 3 we study the problem of *local-implementation* between the κ calculus and $\mathtt{nano}\kappa$ calculus which is a calculus similar to κ that only admits binary interactions and introduces a new kind of rewriting called *bond-flipping*. Roughly speaking a bond-flipping rewriting takes a bond x connecting a molecule A to some other molecule and *migrates* x to a third molecule C concretely connecting C to the molecule that was connected to A through x in the initial state. We give a solution of the local-implementation of κ in $\mathtt{nano}\kappa$ that is divergent and we show the nonexistence of deterministic solutions retaining "reasonable" properties. The results of this chapter have been published in [34].

In Chapter 4 we investigate and compare the expressive power of various calculi within the κ family by studying the leader election problem in a symmetric network. More precisely we compare the calculi by testing their capability in solving the leader election problem in two different scenarios: a fully connected network and a ring network. We work with κ calculus, $\mathtt{nano}\kappa$ calculus and a newly defined κ sub family called $\mathtt{ps}^{\mathrm{n}}$ suitable to describe polymeric structures. In $\mathtt{ps}^{\mathrm{n}}$ by n we represent the maximum number of molecules allowed to react together at once. This family has been defined so that its *less expressive* calculus, i.e. $\mathtt{ps}^2$, be a common subset for both the κ calculus and $\mathtt{nano}\kappa$ calculus. Our investigation led us to realize that the *synchronization degree n*, i.e. the number of elements we allow to communicate simultaneously, has an important role in this context. Our work demonstrate that it is not possible to elect a leader in polymers organized as a suitably large ring. This result entails that polymerizations rewriting at most n proteins are strictly less expressive than those rewriting $n + 1$ proteins. The first scenario was rather unhelpful for any comparison since we prove that the leader election is solvable in all the calculi considered by providing a solving algorithm for the $\mathtt{ps}^2$ calculus: this holds because $\mathtt{ps}^2$ is a subset for both the κ calculus and $\mathtt{nano}\kappa$ calculus, and is also the less powerful representative of the $\mathtt{ps}^\mathtt{n}$ family. The ring network scenario is more interesting. We have that the $\mathtt{ps}^\mathtt{n}$ family is unable to solve the leader election in general but interestingly any given calculus $\mathtt{ps}^\mathtt{m} \in \mathtt{ps}^\mathtt{n}$ can solve the leader election when the ring network size is less than $2m-1$. This is due to the fact that in $\mathtt{ps}^\mathtt{m}$ we can synchronize m elements at a time. A corollary of these result is that the $\mathtt{ps}^\mathtt{n}$ family retains a totally ordered hierarchy wrt the expressive power: $\mathtt{ps}^{\mathtt{m}+1}$ is strictly more powerful than $\mathtt{ps}^\mathtt{m}$. We finally provide two extensions of the common subset $\mathtt{ps}^2$ capable of solving leader election in the ring network. These extensions, $\mathtt{ps}^2_b$ and $\mathtt{ps}^3_c$ are respectively obtained by adding the bond-flipping primitive to the former and allowing bond creation with no constraint in the latter. Since $\mathtt{ps}^2_b$ happens to be a sub calculus for $\mathtt{nano}\kappa$ calculus and $\mathtt{ps}^3_c$ is a sub calculus for κ calculus while the contrary is not true we may conclude that bond-flipping and the free bond creations[1] are the key features for solving the leader election in $\mathtt{nano}\kappa$ calculus and κ calculus respectively. It is worth noticing that the algorithm solving the problem for $\mathtt{ps}^2_b$ does not depend on the network size while all the other considered in this chapter do.

In Chapter 5 we present some purely Theoretical Computer Science results inspired by the previous work in chapters 3 and 4 and specifically by the synchronizations among more than two processes. The idea is to investigate multi synchronization mechanisms, inspired by our work with the κ family, within a well established calculus framework such as the CCS calculus.

We demonstrate the non-existence of a uniform, fully distributed translation of Milner's CCS with multi input synchronizations of $n + 1$ processes into CCS with multi input synchronizations of n processes that retains a "reasonable" semantics. In order to prove this expressiveness gap we adapted a well-known problem of resource sharing: the dining philosophers problem. We define the dining philosophers problem in the hypercube: the philosophers sit at the vertices of an hypercube of dimension n and each edge represents

[1]Together with the ability to involve 3 molecules in a reaction.

a shared fork. We then extend our study to CCS with more powerful synchronizations that allow both inputs and outputs at the same time. We prove that synchronizations with more than three input/output items are encodable in those with three items, while there is an expressiveness gap between three and two. Finally we compare the introduced synchronization mechanisms with various choice operator.

The results of chapters 4 and 5 are in submission process.

Chapter 2

Encoding Simple SA into MDB Brane calculus

In this chapter we provide a translation between two well known families of bio inspired calculi: the P Systems and Brane Calculi families. The translation is rather specific as it involves a couple of calculi within the two families and is not therefore universal nor does equate the two families. Moreover, the translation is one way only as it is shown how to encode Simple SA into MDB but not vice versa. From a purely computational point of view we may consider Simple SA as a lower bound for the computational power of MDB.

It is practically showed that for some system it is possible to shift any formal reasoning on a biological model from one family to the other one.

Membrane systems (also called P systems) and Brane calculi have been recently introduced as formal models inspired by the structure and the functioning of living cells, but having in mind different goals. The aim of Membrane systems was the formal investigation of the computational nature and power of various features of the cell, while Brane calculi aims to define a model capable of a faithful and intuitive representation of various biological processes.

The common background of the two formalisms and the recent growing of interests in applying P systems in Systems Biology have raised the natural question of bridging this two research areas. This chapter goes in this direction, as it presents a direct simulation of a variant of P systems by means of Brane calculi. In particular, we consider a Brane calculus based on three operations called Mate/Bud/Drip, and we show how to use such system to simulate Simple Symport/Antiport P systems, a variant of P systems purely based on communication of objects. As an example, a simplified sodium–potassium pump modeled in Simple SA is encoded in Mate/Bud/Drip Brane calculus.

2.1 Introduction

Both P systems [47] and Brane calculi [13] have been introduced as (families of) computational models inspired by the structure and the functioning of the living cell, starting from the key observation that the various processes taking place in the living cell can be regarded as computations.

Although they have common bases, they have different goals: the main objective of P system's area of research is the formal investigation of the computational nature and power of various features of membranes, while Brane calculi's main goal is to create a model capable of a faithful and intuitive representation of the biological reality.

To better clarify the different goals we quote from [15]:

> "While membrane computing (i.e. P systems) is a branch of natural computing, which tries to abstract computing models, in the Turing sense, from the structure and the functioning of the cell, making use especially of automata, languages, and complexity theoretic tools, Brane calculi pay more attention to the fidelity to the biological reality, have as primary target systems biology, and use especially the framework of process algebra."

Living cells are extremely well organized self-contained systems, composed by discrete interacting components. Each component is delimited by a physical membrane and is often called region or compartment. From a computer science point of view, we can surely regard the cells as information processing devices. A cell has a complex internal activity and an ingeniously elaborated interactions with the environment and the neighbouring cells. A proper interaction based on a flow of substances between components and the environment is necessary for the cell life: a cell not aware of the environment conditions soon dies.

In P systems the cell organization is simulated by a *membrane structure* formalized with a Venn diagram of nested membranes, while the chemicals swimming in the solution delimited by a compartment are represented with a multiset of symbols/objects from a given alphabet. The multiset notion perfectly simulates the unordered structure of floating chemicals.

In the basic variant of P systems the objects evolve according to *evolution rules*, locally associated to the compartments, that transform multisets of symbols in multisets of symbols. The interaction between compartments is realized by evolution rules that move objects between directly nested membranes. Such rules are generally applied non deterministically in a maximally parallel manner: the rules to be used and the objects to evolve are randomly chosen, and, in each step all objects which can evolve must do it.

While in Brane calculi we have a *membrane structure* too, membranes are not simple separators of compartments as in P systems but they are coordinators and active sites of major activity. In Brane calculi a computation happens *on the membrane* and not inside of it. So we no longer make use of multisets of objects but work with processes that reside on membranes. The operations of the two basic Brane calculi proposed in [13] are directly inspired by biologic processes such as endocytosis, exocytosis and mitosis. Another difference with P systems is that generally Brane calculi evolve using an interleaving semantics (sequential single instruction execution).

Only recently P systems have been applied to model biological systems and processes (in particular at cellular level). The common background of the two formalisms and the recent shift of interests of P systems toward Systems Biology have raised the natural question of bridging the two research areas.

Various formal results have already been achieved. In [10] the computational power of two basic Brane calculi proposed in [13] is investigated. In [15] is inspected a variant of P systems inspired by the interactions of a basic Brane calculus defined in [13]. A parallel semantics for Brane calculi, inspired by the maximal parallelism semantics of P systems, is considered in [9]. Finally in [52] two variants of P systems whose interactions are inspired by the two basic Brane calculi defined in [13] are studied.

This chapter follows these recent attempts to connect P systems with Brane calculi, and it contributes with a straight simulation of a variant of P systems by means of Brane calculi. In particular, we consider the variant of Brane calculi based on three operations called Mate/Bud/Drip [13]: such primitives are inspired by membrane fusion (mate) and fission (mito). The fission process is modelled by both *Bud* and *Drip* operations: the former consists in splitting off exactly one internal membrane while the latter in splitting off zero internal membrane.

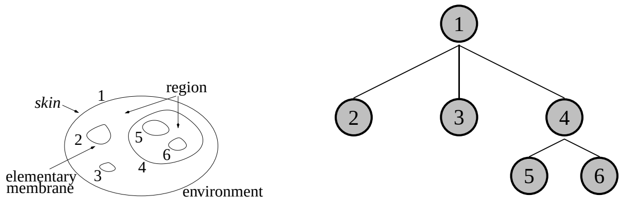
We will show how Mate/Bud/Drip Brane calculus (MBD) can be used to simulate Simple SA systems [29], a bounded variant of P systems with symport/antiport rules [46] where computations are purely based on communication of objects. Such systems are closely related to the biologic trans-membrane communication by coupling chemicals and do not evolve by transforming objects, as the basic variant of P systems, but by changing their position with respect to the compartments defined by the membrane structure used.

The rest of the chapter is organized as follows. In section 2.2 we introduce all the formal notions we will need in the rest of the chapter. Section 2.3 contains the description of the simulation of Simple SA systems in the MDB Brane calculus. In section 2.4 we show an example of a translation: we encode a simplified sodium–potassium pump modeled in Simple SA in MDB Brane calculus. In section 2.5 we report some final remarks and we give some perspectives for future work.

2.2 Preliminaries

The notions of formal language theory we use are basic, and can be found in every monograph in this area (e.g. [58]).

The key function of a biological membrane in a living cell is to define a compartment and its interaction with the surrounding environment (including other compartments). In both P systems and Brane calculi we have a membrane structure where the membrane concept is used as a logic separator between processes and resources. The membrane structure can be seen as a tree like structure, a Venn diagram or a correctly matching parentheses string and it should be clear that we can have nested membrane structures. The most external membrane of a system is called *skin* membrane and a membrane not



$[_1\ [_2]_2\ [_3]_3\ [_4\ [_5]_5\ [_6]_6\ ]_4]_1$

(a) Venn diagram (b) Tree like (c) Matching parentheses

Figure 2.1: Three equivalent ways to represent the same membrane structure

containing other membranes it is said to be *elementary*. The space defined by a membrane is called *region* or compartment. A biologic membrane contains various substances and we use the multiset notion to formalize them. A multiset W over a set X is a mapping $W : X \to \mathbf{N}$ and can be represented with a string (e.g. $a^3bc^2 = abacac = baccaa = \dots$). The multiset data structure let us directly formalize the multiplicities and the unordered structure of floating chemicals.

2.2.1 P systems

P systems aim to abstract new computing paradigms from the biological reality of living cells using tools from the field of automata and formal language theory. Generally speaking a P system is a computational model based on a membrane structure[1] and processes locally multisets of symbols in a parallel and distributed manner. Each membrane identifies a compartment (or region) that *contains* a set of evolution rules and a multiset of symbols from a given alphabet. The symbols in each compartment evolve according to the set of rules contained in it. Typically the rules are used in a non deterministic, maximally parallel way; at each step of computation rules and objects (symbols) are associated in a non deterministic way until no further choice can be made and then the rules are simultaneously applied to the assigned symbols. A computation is successful if it halts and a simple way to define the output is to consider the number of objects in a predefined compartment.

One of the most interesting classes of P systems is the *symport/antiport* one, first

[1]Usually a hierarchical structure as shown in figure 2.1, but a net structure exists too.

introduced in [46]. P systems with symport/antiport rules (briefly S/A P systems) are closely related to the biological trans-membrane communication that occurs at the cellular level and relies on coupling chemicals. When two chemicals can cross a membrane only together in the same direction we speak of *symport* process; when the two chemicals can cross a membrane only together but in opposite direction we speak of *antiport* process. Given an alphabet of objects V and objects $a, b \in V$ we formalize these processes with rules of the form (ab, in) or (ab, out) for symport and $(a, out; b, in)$ for antiport. We also consider rules of the form (a, in) or (a, out) and call them *uniport* rules.

A system with only these rules is *purely communicative* since no objects are destroyed or transformed during the computation, they only change the compartment they belong to. Formally a S/A P system is a construct

$$\Pi = (V, \mu, W_1, \ldots, W_m, E, R_1, \ldots, R_m, i_0)$$

where: V is the alphabet of objects; μ is a membrane structure with m membranes injectively labelled by positive integers $1, 2, \ldots, m$; $W_1, \ldots, W_m$ are the multisets of objects (symbols) initially contained in the compartments; $E \subseteq V$ is the set of objects present in arbitrarily many copies in the environment; $R_1, \ldots, R_m$ are finite sets of rules associated with the m membranes of μ; $i_0 \in \{1, \ldots, m\}$ is an elementary membrane of μ (the output membrane). The *configuration* of a S/A P system is defined by the multisets of objects present in all the regions of Π. The result of a S/A P system is the number of objects present in the membrane (compartment) labelled i_0 in the halting configuration.

Simple SA

A Simple SA[2] (of degree $k + 1$) is a limited (i.e. non universal) variant of a S/A P system, and is defined by a construct

$$\Pi = (V, \mu, W_1, \ldots, W_{k+1}, W_E, n_1, \ldots, n_k, R_1, \ldots, R_{k+1})$$

where

- the alphabet is $V = F \cup \{o\}$ with F finite set of symbols and $o \notin F$;

[2]Obviously “SA” stands for “symport/antiport”.

- μ is a two level membrane structure: $k \geq 1$ *input membranes* on the same level embedded in a *skin membrane* $\rightarrow [_{skin}[_1]_1 \ldots [_k]_k]_{skin}$;
- W_i, W_E are finite multisets of objects over F respectively associated with the region i and the environment;
- n_i is the (non negative) number of objects o initially contained in the input membrane i (the skin membrane initially does not contain any o);
- R_i is a finite set of symport/antiport rules associated to the region i. They are of the form

  $$\cdot\text{(a) } (u, out) \qquad \cdot \text{ (b) } (v, in) \qquad \cdot \text{ (c) } (u, out; v, in)$$

  with $u, v \in V^+$, and for rules of type (b) and (c) it holds $o \notin v$: objects o can "move" only toward the environment region.

In [29] Simple SA are studied as acceptors of k-tuples $(n_1, \ldots, n_k)$ of non negative integers. A k-tuple is accepted if, when the k input membranes are given $o^{n_1}, \ldots, o^{n_k}$, the system halts. It is shown that Simple SA systems characterize exactly the semilinear sets [28].

Compared with basic S/A P systems, Simple SA is formally a less powerful device but, from a biologic point of view, the limited resources scenario of Simple SA makes it a more realistic model for Systems Biology. Standard S/A P systems have indeed uncountable many copies of each symbol in the environment while obviously biologic systems have to cope with limited resources.

2.2.2 Brane calculi

Brane calculi are a family of process calculi proposed for modelling the various cell dynamics together with the behaviour of biological membranes.

The main difference with regard to P systems is that here active entities are tightly coupled to membranes and this endows membranes with the (correct) biological role of support of biochemistry; we have that computation happens on membrane and not inside of it. Another difference with P systems is that generally Brane calculi evolve using an interleaving semantic (sequential single instruction execution).

Biologic membranes are formed by a lipid bilayer that actually behaves as a fluid that let both structural components and embedded substances like proteins to freely move into the region delimited by the lipid layers. Membranes themselves are immersed into an aqueous solution where they can freely float. Such a *fluid within fluid* structure directly inspires the basic structure of Brane calculi: two commutative monoids with a replication operator, each representing a kind of fluid. We have

1. a monoid representing the lipid bilayer: $(Membranes, |, 0)$
   where | is the membranes' composition and 0 the unit (empty process);

2. and a monoid representing the aqueous solution: $(Systems, \circ, \diamond)$
   where $\circ$ is the systems' composition and $\diamond$ the unit (empty system).

Both monoids use the replication operator ! to model the notion of a "multitude" of components of the same type (parallel composition of an unbounded number of components).

A system consists of nested membranes and each membrane represents a combination of actions (a process) that define its behaviour. Formally the basic syntax is described by the following table

Systems	P,Q ::= $\diamond$ ¦ P $\circ$ Q ¦ !P ¦ $\sigma\langle P\rangle$	nested membranes
Membranes	σ, τ ::= 0 ¦ $\sigma|\tau$ ¦ $!\sigma$ ¦ $a.\sigma$	membrane processes
Actions	a, b ::= ...	see table 2.4

Table 2.1. Basic syntax of Brane calculi

With $\sigma\langle P\rangle$ we denote a generic system that behaves as σ and contains the system P. With $a.\sigma$ we denote a guarded process: the process behaves as σ after the execution of the action a. We use the following abbreviations: a for $a.0$, $\langle P\rangle$ for $0\langle P\rangle$, and $\sigma\langle\rangle$ for $\sigma\langle\diamond\rangle$. Both systems and processes have a structural congruence relation defined over them (table 2.2).

$P \circ Q \equiv Q \circ P$	$\sigma \vert \tau \equiv \tau \vert \sigma$
$P \circ (Q \circ R) \equiv (P \circ Q) \circ R$	$\sigma \vert (\tau \vert \rho) \equiv (\sigma \vert \tau) \vert \rho$
$P \circ \diamond \equiv P$	$\sigma \vert 0 \equiv \sigma$
$!\diamond \equiv \diamond$	$!0 \equiv 0$
$!(P \circ Q) \equiv !P \circ !Q$	$!(\sigma \vert \tau) \equiv !\sigma \vert !\tau$
$!!P \equiv !P$	$!!\sigma \equiv !\sigma$
$!P \equiv P \circ !P$	$!\sigma \equiv \sigma \vert !\sigma$
$0 \langle \diamond \rangle \equiv \diamond$	

Table 2.2. Structural congruence relation for systems and membranes

The generic reactions of table 2.3 are valid for any Brane calculus, but each calculus have to define its own specific reactions providing the relative reaction rules.

$$P \Rightarrow\!\!\!\!\rightarrow Q \Rightarrow P \circ R \Rightarrow\!\!\!\!\rightarrow Q \circ R$$

$$P \Rightarrow\!\!\!\!\rightarrow Q \Rightarrow \sigma \langle P \rangle \Rightarrow\!\!\!\!\rightarrow \sigma \langle Q \rangle$$

$$P \equiv P' \wedge P' \Rightarrow\!\!\!\!\rightarrow Q' \wedge Q' \equiv Q \Rightarrow P \Rightarrow\!\!\!\!\rightarrow Q$$

We use $\Rightarrow\!\!\!\!\rightarrow^*$ to denote the transitive and reflexive closure of $\Rightarrow\!\!\!\!\rightarrow$[3].

Table 2.3. Generic reaction rules

The Mate/Bud/Drip Brane calculus

The Mate/Bud/Drip Brane calculus (briefly MDB) is defined by three actions inspired by the biological processes of membrane fusion (mate) and fission (mito). The fission process is specialized into two operations: *budding* (bud), which consists in the splitting off one internal membrane, and *dripping* (drip), which consists in the splitting off zero internal membranes. The fusion process is represented by the *mating* (mate) operation.

Tables 2.4 and 2.5 provide the MDB actions' syntax and reaction rules.

Actions	a,b ::= $\mathsf{mate}_n \mid \mathsf{mate}_n^{\perp} \mid \mathsf{bud}_n \mid \mathsf{bud}^{\perp}(\rho) \mid \mathsf{drip}(\rho)$

Table 2.4. MDB actions' syntax

Mate	$\mathsf{mate}_n\, \sigma\vert\sigma_0\, \langle P\rangle \circ \mathsf{mate}_n^{\perp}\, .\tau\vert\tau_0\, \langle Q\rangle \Rightarrow \sigma\vert\sigma_0\vert\tau\vert\tau_0\, \langle P \circ Q\rangle$
Bud	$\mathsf{bud}_n^{\perp}(\rho).\tau\vert\tau_0\, \langle \mathsf{bud}_n\, .\sigma\vert\sigma_0\, \langle P\rangle \circ Q\rangle \Rightarrow \rho\, \langle\sigma\vert\sigma_0\, \langle P\rangle\rangle \circ \tau\vert\tau_0\, \langle Q\rangle$
Drip	$\mathsf{drip}(\rho).\sigma\vert\sigma_0\, \langle P\rangle \Rightarrow \rho\, \langle\rangle \circ \sigma\vert\sigma_0\, \langle P\rangle$

Table 2.5. Reaction rules for MDB

For actions that involve two membranes we need a co-action to identify the second membrane. Such co-actions are obtained appending the symbol $^{\perp}$ to the action name. Moreover we can index our actions in order to precisely couple an action with the correct co-action.

Actions mate_n and $\mathsf{mate}_n^{\perp}$ will synchronize to achieve membranes fusion and the membranes will result irreversible mixed. Actions bud_n and $\mathsf{bud}_n^{\perp}$ will synchronize to split one nested membrane. Action drip let to split off zero internal membranes. Actions drip and $\mathsf{bud}^{\perp}$ come with a parameter ρ which represents the membrane process of the new membrane created by the reaction.

2.2.3 MDB syntax simplifications

Let k be a positive integer and ω a membrane process (combinations of actions, see table 2.1), we use the following syntax simplifications

[3]Roughly speaking this means that the reaction is not an elementary transition but is composed by a sequence of elementary reactions.

$$k(\omega) \triangleq \overbrace{\omega.\omega.\ldots.\omega}^{k} \text{ and } (\omega)^k \triangleq \overbrace{\omega|\omega|\ldots|\omega}^{k}$$

Moreover, where clear enough, we often omit the parentheses and use just $k\omega$ and ω^k. With indexed membrane process we use $\prod$ to denote the parallel composition

$$\prod_{i=1}^{k} \omega_i \triangleq \omega_1 \mid \ldots \mid \omega_k$$

and the symbol $\sum$ for

$$\sum_{i=1}^{k} \omega_i \triangleq \omega_1.\omega_2.\ldots.\omega_k$$

Similarly for parallel composition of k identical systems we use

$$(P)^k \triangleq \overbrace{P|P|P|P}^{k}$$

where P is a system.

2.2.4 Synchronization actions

In order to deal with common synchronization situations for adjacent and directly nested membranes in the MDB calculus, we introduce the composite synchronization actions syn and $\mathsf{syn}^{\perp}$.

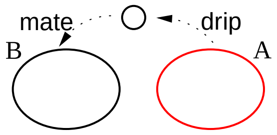
Let's assume we want to model a dependency of process ω_{B_1} from process ω_{A_1}. More generally let ω_{B_1} and ω_{A_1} be two processes that need to synchronize, with ω_{B_1} the process waiting for a signal from ω_{A_1}. Clearly ω_{A_1} and ω_{B_1} must be in parallel.

Although syn and $\mathsf{syn}^{\perp}$ are designed as an *action - coaction* couple, they, for their composite nature, do not happen simultaneously. A syn action is free to execute whenever able to, while the corresponding $\mathsf{syn}^{\perp}$ action can execute only if the former already executed, and between the two, arbitrarily many actions can execute. We define the following systems

$$\begin{aligned} \mathrm{A} &\triangleq \omega_{A_1}.\mathsf{syn}_{AB} \mid \omega_{A_2} \langle \ldots \rangle \\ \mathrm{B} &\triangleq \mathsf{syn}^{\perp}_{AB}.\omega_{B_1} \mid \omega_{B_2} \langle \ldots \rangle \end{aligned} \tag{2.1}$$

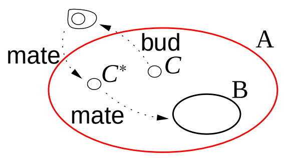
For greater flexibility, we define the synchronization actions syn and $\mathsf{syn}^\perp$ as special contextual operations. That means they have different definitions determined by the context (i.e. the membranes involved) they are used in. While modelling, we can simply regard syn as the "send signal" action and $\mathsf{syn}^\perp$ as the "receive signal" action, but it should be clear that we are effectively dealing with three different couples of definitions. We indeed have the following scenarios:

1. *A* and *B* adjacent:



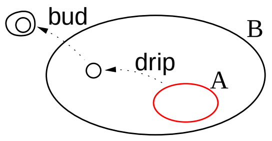
$$\begin{cases} S \triangleq \omega_{A_1}.\mathsf{syn}_{AB}\langle\ldots\rangle \circ \mathsf{syn}^\perp_{AB}.\omega_{B_1}\langle\ldots\rangle \\ \mathsf{syn}_{AB} \triangleq \mathsf{drip}(\mathsf{mate}_{AB}) \\ \mathsf{syn}^\perp_{AB} \triangleq \mathsf{mate}^\perp_{AB} \end{cases}$$

2. *B* nested in *A*:



$$\begin{cases} S \triangleq \omega_{A_1}.\mathsf{syn}_{AB}\Big\langle C \circ \overbrace{\mathsf{syn}^\perp_{AB}.\omega_{B_1}\langle\ldots\rangle}^{B}\Big\rangle \\ \mathsf{syn}_{AB} \triangleq \mathsf{bud}^\perp_{AB}(\mathsf{mate}_{AB}) \mid \mathsf{mate}^\perp_{AB} \\ C \triangleq \mathsf{bud}_{AB}.\mathsf{mate}_{AB}\langle\rangle \\ \mathsf{syn}^\perp_{AB} \triangleq \mathsf{mate}^\perp_{AB} \end{cases}$$

3. *A* nested in *B*:



$$\begin{cases} S \triangleq \mathsf{syn}^\perp_{AB}.\omega_{B_1}\Big\langle \overbrace{\omega_{A_1}.\mathsf{syn}_{AB}\langle\ldots\rangle}^{A}\Big\rangle \\ \mathsf{syn}_{AB} \triangleq \mathsf{drip}(\mathsf{bud}_{AB}) \\ \mathsf{syn}^\perp_{AB} \triangleq \mathsf{bud}^\perp_{AB}(0) \end{cases}$$

The further scenario of processes ω_{A_1} and ω_{B_1} on the same membrane in parallel composition is analogous to the scenario of adjacent membranes: the couple syn $\mathsf{syn}^\perp$ is defined the same way, and so a signal syn can be received indistinctly by an adjacent membrane or by a process on the same membrane. We will use this property later in our encoding.

The scenario (2) requires an *ad hoc* support system C: if the syn action is thought to repeat system C must be present with the proper multiplicity (e.g. $!\,\mathsf{syn} \Rightarrow\, !C$).

We leave it as an exercise to verify that in all the scenarios considered above the system S can execute ω_{B_1} only after executing ω_{A_1}. That realizes our dependency:

$$\ldots \Rightarrow \omega_{A_1} \Rightarrow \ldots \Rightarrow \omega_{B_1} \ldots$$

2.3 The encoding

We now show how to construct a non deterministic simulation of Simple SA with MDB calculus. The encoding we propose is non deterministic in the following sense: the simulation behaves non deterministically only when the original (simulated) Simple SA system shows non deterministic behaviour too.

Let Π be the Simple SA system we want to simulate, we call H the set of all available objects[4] in Π. We have that

$$|H \models h = \sum_{j=1}^{k+1} |W_j| + |W_E| + \sum_{j=1}^{k} |o^{n_j}| \tag{2.2}$$

and moreover

1. H is finite;

2. and its cardinality h is fixed for the entire computation.

The first property is the main characteristic of Simple SA systems and it derives from the *finiteness* of the multiset W_E initially given to the environment E (in a basic S/A P system the environment has an *unbounded* quantity of specified objects). The second property is a direct consequence of the intrinsic *conservation law* of S/A P systems: no object is created or destroyed during the computation. From now on with h we always refer to the fixed number of objects of the Simple SA we are considering.

It is now easy to formulate a key observation: in a single step of computation a Simple SA can execute at most h reactions. Indeed, in order to maximize the number of rules

[4]Including the environment objects.

used having exactly h objects, we can (if all conditions are met) choose for execution h uniport rules (they "use" only a single object). It is obvious that after choosing h uniport rules we use all our objects and no more choices are possible. We recall that each region/membrane[5] is identified by a unique index i, and with a little abuse of notation we can identify the environment with the index $k + 2$. We also label the rules by a given set of integers in a one to one manner.

The most relevant parts of our encoding are:

- the (exact) simulation of a non deterministic maximally parallel step of execution;
- the processes that determine if a certain rule i is applicable or not;
- the way we encode the objects and the membrane structure of Π.

2.3.1 Encoding objects and structure

Given a multiset W_l and an object $a \in W_l$ we encode a with the following system

$$\mathsf{mate}_{a,l} \langle\rangle$$

This system models an instance of object a contained by the membrane l. Hence, if $W_l = \alpha_1, \ldots, \alpha_s$, we model W_l with the following composition of systems

$$W_l = \mathsf{mate}_{\alpha_1,l} \langle\rangle \circ \ldots \circ \mathsf{mate}_{\alpha_s,l} \langle\rangle$$

In a similar way we model each o symbol contained by a membrane l with a system

$$\mathsf{mate}_{o,l} \langle\rangle$$

Given a configuration of Π we can easily encode every object in the MDB calculus using the above suggestions obtaining h systems (one for each object of Π) of the form $\mathsf{mate}_{\beta,i} \langle\rangle$ with $\beta \in V$ and $i \in 1, \ldots, k + 2$. It should be clear enough that a composition of all such systems embedded in a single membrane would preserve all the informations

[5]Clearly the correspondence region - membrane is one to one.

of the given Π configuration (the symbols along with their position in the membrane structure).

Clearly two objects of the same type generate two different systems if they belong to different regions. Since we have $k + 2$ regions (we include the environment region) each object $a \in V$ can map in $k + 2$ different systems according to its actual region. With such encoding we effectively collapse the two level structure of Π by expanding the alphabet used by exactly a $k + 2$ factor. In MDB we no longer work with objects from V but with objects from a new alphabet $\dot{V}$ where

$$\dot{V} = \{1, 2, \ldots, k + 2\} \times V$$

It should be clear that each symbol of $\dot{V}$ has a direct interpretation as an MDB system; a bijection that

$$\forall \alpha \in \dot{V} \quad \alpha \longmapsto \mathsf{mate}_\alpha \langle\rangle \tag{2.3}$$

It is worth noting that we can similarly collapse a structure of any complexity with the only constraint of using an accordingly expanded alphabet. Collapsing a structure of degree m would expand the original alphabet V to an alphabet of cardinality $|V|(m + 1)$. We use $\dot{H}$ to denote the set of all the encoded objects of Π. For bijection (2.3) we can interchangeably consider $\dot{H}$ as composed by symbols from $\dot{V}$ or by systems of the form $\mathsf{mate}_\alpha \langle\rangle$ with $\alpha \in \dot{V}$.

2.3.2 Simulating rules

Having encoded the *data structure* we now show how to simulate the symport/antiport rules of Π. Let i be a generic rule associated to membrane l, it can be executed only if all the necessary objects are simultaneously available.

In Simple SA, the verification of the availability of objects is directly demanded to the semantics used, while in Brane calculi we must simulate the verifying process. The only way we can assure the presence of an object α in region l is to execute the action $\mathsf{mate}^{\perp}_{\alpha,l}$ effectively "consuming" the system $\mathsf{mate}_{\alpha,l} \langle\rangle$ that encodes the object.

Since rule i generally requires a string $u \in V^*$ the verifying process for i is a sequence of $\mathsf{mate}^\perp$ actions. For example, let $u = \alpha_1, \ldots, \alpha_n$ and $i = (u, out)$, we say that i is executable if we can execute the following sequence:

$$\mathsf{mate}^\perp_{\alpha_1,l} . \mathsf{mate}^\perp_{\alpha_2,l} . \ldots . \mathsf{mate}^\perp_{\alpha_n,l} \tag{2.4}$$

If the sequence (2.4) completes successfully we can proceed with execution of rule i, but if some object is missing the sequence halts in a *deadlock* condition. To prevent a global deadlock we compose such process with its counterpart: a process that successfully completes if and only if there are not all the necessary objects to execute i. These two process are complementary: if one successfully completes the other will certainly deadlock and vice versa. We call *run* the process that verifies the presence of the necessary objects and then executes the rule i, while we call *skip* the process that verifies the unavailability of the necessary objects. Composing in parallel the two processes we can deterministically establish if rule i can be executed or not. Before detailing the two processes we must introduce the concept of *complementary symbols/objects*.

Complementary objects

We have already suggested a way to evaluate the executability of a rule (i.e. the equation (2.4)) but not how to verify the contrary (the *skip* process). Let

$$\lambda_{i,1}^{m_{i,1}} \lambda_{i,2}^{m_{i,2}} \ldots \lambda_{i,t_i}^{m_{i,t_i}}, \quad \lambda_{i,j} \in \dot{V}, m_{i,k} \in \mathbf{N}^+$$

be the multiset of objects required by rule i. If we denote with $|\dot{H}|_{\lambda_{i,j}}$ the number of occurrences of system $\mathsf{mate}_{\lambda_{i,j}} \langle\rangle$ in set $\dot{H}$ the followings holds

$$|\dot{H}|_{\lambda_{i,1}} \geq m_{i,1} \wedge |\dot{H}|_{\lambda_{i,2}} \geq m_{i,2} \wedge \cdots \wedge |\dot{H}|_{\lambda_{i,t_i}} \geq m_{i,t_i} \Leftrightarrow \text{ rule } i \text{ is applicable} \tag{2.5}$$

$$|\dot{H}|_{\lambda_{i,1}} < m_{i,1} \vee |\dot{H}|_{\lambda_{i,2}} < m_{i,2} \vee \cdots \vee |\dot{H}|_{\lambda_{i,t_i}} < m_{i,t_i} \Leftrightarrow \text{ rule } i \text{ is } \textbf{not} \text{ applicable} \tag{2.6}$$

We are interested to, and will use, only the ($\Rightarrow$) part of the above equations. The interpretation of (2.5) is quite straightforward: if the multiplicities of objects required by rule i are *less or equal* to the corresponding multiplicities of objects present in $\dot{H}$ then rule i can be executed. We already know we can verify the antecedent of (2.5) with the following sequence

$$m_{i,1}\left(\mathsf{mate}^{\perp}_{\lambda_{i,1}}\right).m_{i,2}\left(\mathsf{mate}^{\perp}_{\lambda_{i,2}}\right).\dots.m_{i,t_i}\left(\mathsf{mate}^{\perp}_{\lambda_{i,t_i}}\right) \tag{2.7}$$

In order to verify the antecedent of (2.6) we define the alphabet of the complementary objects of $\dot{V}$

$$C\left(\dot{V}\right) = \{\overline{\alpha} : \alpha \in \dot{V}\}$$

If we add to $\dot{H}$ systems from $\left\{\mathsf{mate}_{\overline{\alpha}}\langle\rangle : \ \overline{\alpha} \in C\left(\dot{V}\right)\right\}$ in such a way that

$$|\dot{H}|_{\alpha} + |\dot{H}|_{\overline{\alpha}} = h \ \ \forall\alpha \in \dot{V} \tag{2.8}$$

we have that $|\dot{H}|_{\overline{\alpha}}$ corresponds exactly to the number of systems in $\dot{H}$ different from $\mathsf{mate}_{\alpha}\langle\rangle$ (not counting the complementary systems!). Hence we can write the following equivalence

$$\begin{aligned} &|\dot{H}|_{\lambda_{i,1}} < m_{i,1} \vee |\dot{H}|_{\lambda_{i,2}} < m_{i,2} \vee \cdots \vee |H|_{\lambda_{i,t_i}} < m_{i,t_i} = \\ &|\dot{H}|_{\overline{\lambda_{i,1}}} > h - m_{i,1} \vee |\dot{H}|_{\overline{\lambda_{i,2}}} > h - m_{i,2} \vee \cdots \vee |\dot{H}|_{\overline{\lambda_{i,t_i}}} > h - m_{i,t_i} \end{aligned} \tag{2.9}$$

Using (2.9) we can now verify the antecedent of (2.6) with the following composition of processes

$$h_1\left(\mathsf{mate}^{\perp}_{\overline{\lambda_{i,1}}}\right) \Big| \ h_2\left(\mathsf{mate}^{\perp}_{\overline{\lambda_{i,2}}}\right) \Big| \dots \Big| \ h_{t_i}\left(\mathsf{mate}^{\perp}_{\overline{\lambda_{i,t_i}}}\right) \tag{2.10}$$

where $h_j = (h - m_{i,j} + 1)$. More precisely the antecedent is verified if any of the t_i processes in the above composition successfully complete. For the sake of clarity, we observe that in (2.10) we use parallel composition while in (2.7) we use a straight sequence because of the different "*normal forms*[6]" of the two logic expressions (2.6) and (2.5): the former is true *if any* ..., while the latter *if all* .... For that reason we will later use the $\sum$ operator for the *run* process definition and the $\prod$ operator for the *skip* one.

Effect of an antiport rule

As an example we show the effect of applying to our encoded objects an antiport rule. Let i be now an antiport rule $(u, out; v, in)$ associated to membrane s. Given

$$u \triangleq \alpha_1 \dots \alpha_l \ \text{ and } \ v \triangleq \beta_1 \dots \beta_e \ \text{ with } \ \alpha_j \in V, \beta_j \in F$$

[6] Yes, we are abusing the "conjunctive and disjunctive normal forms" definitions of boolean expressions!

if set $\dot{H}$ contains systems $\overleftrightarrow{\mathsf{mate}_{\alpha_{1,s}} \langle \diamond \rangle, \dots, \mathsf{mate}_{\alpha_{l,s}} \langle \diamond \rangle}^{u}$ encoding string u, and systems $\overleftrightarrow{\mathsf{mate}_{\beta_{1,k+1}} \langle \diamond \rangle, \dots, \mathsf{mate}_{\beta_{e,k+1}} \langle \diamond \rangle}^{v}$ encoding string v than, for (2.5), we can execute rule i and its execution would have the following effect on the system

$$\begin{aligned} \mathsf{mate}_{\alpha_{j,s}} \langle \diamond \rangle &\overset{rule-i}{\Rightarrow} \mathsf{mate}_{\alpha_{j,k+1}} \langle \diamond \rangle, \quad \forall j : 1 \le j \le l \\ \mathsf{mate}_{\beta_{j,k+1}} \langle \diamond \rangle &\overset{rule-i}{\Rightarrow} \mathsf{mate}_{\beta_{j,s}} \langle \diamond \rangle, \quad \forall j : 1 \le j \le e \end{aligned} \tag{2.11}$$

where $k + 1$ is the index identifying the *skin* region. Formally, in our simulation, a rule i transforms a string γ_i on $\dot{V}$ into another string ω_i on $\dot{V}$ (i.e. $\gamma_i \overset{i}{\longmapsto} \omega_i$). We define, for later use, γ_i and ω_i as

$$\gamma_i = \gamma_{i,1}\gamma_{i,2} \dots \gamma_{i,x_i}, \ \omega_i = \omega_{i,1}\omega_{i,2} \dots \omega_{i,x_i} \quad \text{with } x_i = |uv| = l + e$$

Having $|\gamma_i \models |\omega_i|$ means our encoding respects the conservation law observed by Simple SA.

2.3.3 Simulating rules (encoding)

We finally show and explain the concrete encoding of the processes *run* and *skip*. Let's suppose we are evaluating the executability of our antiport rule i: we have process run_i and process $skip_i$ in parallel execution on distinct membranes.

Although in theory *run* and *skip* are mutually exclusive (if one successfully complete the other will not), we still have a *race condition* (see below) and we resolve it by modeling a (unique) system $\mathsf{mate}_{Token} \langle \rangle$; the first process that successfully completes consumes the *token* performing the action $\mathsf{mate}^{\perp}_{Token}$. We remark that for every couple of processes *run* and *skip* there is a single *token*, and the *token* itself is responsible for the entire synchronization between the two processes.

Since both processes effectively modify the configuration of the global system by consuming objects during their verification, it may happen that, for example, run_i blocks (i.e. fails) after consuming some objects. In such case we must restore, with a proper *undo* process, the consumed objects but do not know exactly which objects were consumed. We solve this by first introducing in $\dot{H}$ all the objects required by run_i and then waiting for its completion. In this way, we induce a *fake* termination of run_i and, as a net result, we restore the configuration prior to the execution of run_i itself. The undo process for $skip_i$ is solved in the same way. It is the fake termination that cause the race condition.

Clearly there are two cases:

1. i is executable $\Rightarrow$ process run_i complete successfully;

2. i is not executable $\Rightarrow$ process $skip_i$ complete successfully.

We separately describe the two situations.

The "run" process

$$\begin{aligned} run_i \triangleq {} & \Big(\sum_{p=1}^{x_i} \big(\mathsf{mate}^{\perp}_{\gamma_{i,p}}\big) . \mathsf{syn}_A . \mathsf{mate}^{\perp}_{Token} . \mathsf{syn}^{\perp}_A . undo\text{-}skip_i . \mathsf{syn}_{TrashSkip} . \\ & . \mathsf{syn}^{\perp}_{TrashDone} . inc\text{-}\overline{\gamma_i} . \mathsf{drip}(rule_i) . \mathsf{drip}\left(\mathsf{syn}_{TrashRun} . \mathsf{syn}^{\perp}_{TrashDone} . \mathsf{syn}_{Next}\right)\Big) \\ & \mid \mathsf{syn}^{\perp}_{TrashRun} . \mathsf{bud}_{Junk} \\ rule_i \triangleq {} & \mathsf{syn}^{\perp}_{Execute} . \sum_{j=1}^{x_i} \big(\mathsf{drip}(\mathsf{mate}_{\omega_{i,j}})\big) . dec\text{-}\overline{\omega_i} . \mathsf{syn}_{Executed} \end{aligned} \tag{2.12}$$

Here we assume having all the necessary objects to execute i, and therefore to successfully complete run_i. Briefly the run_i process uses up, by performing the sequence of actions $\sum_{p=1}^{x_i}\big(\mathsf{mate}^{\perp}_{\gamma_{i,p}}\big)$, the objects it requires, then it consumes the *token* with the $\mathsf{mate}^{\perp}_{Token}$ action and finally it undoes any effect on the system caused by the $skip_i$ process (i.e. any action done by $skip_i$) by executing the process $undo\text{-}skip_i$ (detailed later). The action $\mathsf{syn}_{TrashSkip}$ orders the "expulsion" (i.e. the *budding* to the environment region) of the $skip_i$ membrane from the system (if left it can adversely affect the simulation): $\mathsf{syn}^{\perp}_{TrashDone}$ action couples with *skin*'s $\mathsf{syn}_{TrashDone}$ action, and they're executed only after $skip_i$ membrane has been *trashed*. Actually $\mathsf{syn}_{TrashDone}$ behaves as an acknowledgement signal. The process $inc\text{-}\overline{\gamma_i}$ releases a system $\mathsf{mate}_{\overline{\gamma_{i,j}}}\langle\rangle$ for each object $\gamma_{i,j}$ previously consumed by run_i. We remind that we work with a fixed number of objects h, hence for (2.8) by creating a new system $\mathsf{mate}_{\overline{\gamma_{i,j}}}\langle\rangle$ in $\dot{H}$ we are effectively informing the simulation that the number of available objects $\gamma_{i,j}$ is decremented by one.

Up to now we have only reserved the required objects and modified accordingly the complementary ones. The process $rule_i$ contains the sequence that actually executes the rule by creating the objects composing the string ω_i but, after being released, $rule_i$ remains idle until it receives the signal $\mathsf{syn}_{Execute}$ (detailed later). The $dec\text{-}\overline{\omega_i}$ process is analogous but contrary to process $inc\text{-}\overline{\gamma_i}$ seen before: it consumes a system $\mathsf{mate}_{\overline{\omega_{i,j}}}\langle\rangle$ for each $\omega_{i,j} \in \omega_i$ created with $\sum_{j=1}^{x_i}(\mathsf{drip}(\mathsf{mate}_{\omega_{i,j}}))$.

As for $skip_i$, if we leave the remainder of the run_i process it can adversely affect the simulation, so we release the process $\mathsf{syn}_{TrashRun} \,.\, \mathsf{syn}^{\perp}_{TrashDone} \,.\, \mathsf{syn}_{Next}$ that first causes the expulsion (budding) to the environment region of run_i membrane itself and then signals the correct termination with the action syn_{Next}.

The "skip" process

$$\begin{aligned} skip_i \triangleq \prod_{j=1}^{t_i}\Big(& h_j\, \mathsf{mate}^{\perp}_{\overline{\lambda_{i,j}}} \,.\, \mathsf{syn}_B \,.\, \mathsf{mate}^{\perp}_{Token} \,.undo\text{-}skip_i.undo\text{-}run_i.\, \mathsf{syn}_{TrashRun} \,. \\ & .\, \mathsf{syn}^{\perp}_{TrashDone} \,.\, \mathsf{drip}\left(\mathsf{syn}_{TrashSkip} \,.\, \mathsf{syn}^{\perp}_{TrashDone} \,.\, \mathsf{syn}_{Fail}\right)\Big) \mid t_i\, \mathsf{syn}^{\perp}_{B} \,.\, \mathsf{syn}_{UndoDone} \\ & \mid\ \mathsf{syn}^{\perp}_{TrashSkip} \,.\, \mathsf{bud}_{Junk} \end{aligned} \tag{2.13}$$

In brief, $skip_i$ verifies that process run_i is not going to complete successfully, then restores the complementary objects it used for verification performing process $undo\text{-}skip_i$ and undoes the effect of the run_i process too executing the process $undo\text{-}run_i$. Finally it forces the expulsion of both run_i and $skip_i$ membranes and signals to the *controller* (see (2.15)) that i is not applicable with the action syn_{Fail}.

The $skip_i$ process establishes the unexecutability of rule i (i.e. the failure of run_i) if any of the t_i parallel processes it is composed of reaches the execution of the action syn_B. Such a process consumes the *token* and follows as described above. Looking at (2.13) you should understand that each of the t_i parallel processes verifies the **unavailability** of the corresponding $\lambda_{i,j}$ object. That means that if the process indexed by $j = 3$ executes action syn_B the number of objects $\lambda_{i,3}$ in the system is not sufficient for the execution of $rule_3$.

After the execution of the sequence $undo\text{-}skip_i.undo\text{-}run_i$ the set $\dot{H}$ returns to the state prior to the parallel execution of run_i and $skip_i$. If a rule can not execute it clearly should not use any object.

The signal $\mathsf{syn}_{UndoDone}$ is sent only if all the t_i parallel processes have executed their syn_B actions; the *undo-skip*$_i$ process enforces that by releasing exactly the systems $\mathsf{mate}_{\overline{\lambda_{i,j}}}\langle\rangle$ needed to partially complete all the t_i parallel processes *skip*$_i$ is composed of. We point out that only one of the t_i processes will perform actions that follow syn_B since there is only one system $\mathsf{mate}_{Token}\langle\rangle$ available. Moreover this will occur only if it is impossible to complete process *run*.

$$
\begin{aligned}
\mathit{undo\text{-}run}_i &\triangleq \Big(\sum_{j=1}^{x_i} \mathsf{drip}\big(\mathsf{mate}_{\gamma_{i,j}}\big)\Big).\,\mathsf{syn}_A^{\perp} \\
\mathit{undo\text{-}skip}_i &\triangleq \Big(\sum_{j=1}^{t_i} h_j\,\mathsf{drip}\big(\mathsf{mate}_{\overline{\lambda_{i,j}}}\big)\Big).\,\mathsf{syn}_{UndoDone}^{\perp} \\
\mathit{inc}\text{-}\overline{\gamma_i} &\triangleq \sum_{j=1}^{x_i} \mathsf{drip}\big(\mathsf{mate}_{\overline{\gamma_{i,j}}}\big) \\
\mathit{dec}\text{-}\overline{\omega_i} &\triangleq \sum_{j=1}^{x_i} \mathsf{mate}_{\overline{\omega_{i,j}}}^{\perp} \\
\mathit{token} &\triangleq \mathsf{mate}_{Token}
\end{aligned}
\tag{2.14}
$$

2.3.4 Simulating the non deterministic maximally parallel step

To faithfully simulate the Simple SA parallel semantics we must first non deterministically determine a maximal multiset of applicable rules and then simultaneously apply them. The number of times a particular rule is executed in a specific step is not bounded, but we recall that we can execute no more than h rules at any step of computation. Given a configuration there may be different maximal multisets of rules, the Simple SA semantics non deterministically selects one.

Since MDB uses a sequential semantics, we must be able to non deterministically choose up to h rules one by one. We realize this by means of systems Lv: the role of a generic system Lv_j, with $1 \leq j \leq h$, is to select (if possible) the j–th rule and add it to the maximal set in construction. If system Lv_j is active, then it means we have already chosen for execution $j-1$ rules. Only one Lv system is active at a time.

Let r be the number of rules in Π. The simulation of a step of computation initiates by sending the signal syn_1 and activating the various processes of system Lv_1.

When a generic Lv_j system activates, its *controller* process sends a syn_{Start} signal that non deterministically activates one of the processes that compose the process *rules*. Process *rules* is composed by exactly r parallel processes, one for each rule of Π. Let i be the index of the activated process: it releases systems $run_i\langle\rangle$, $skip_i\langle\rangle$ and $token\langle\rangle$. The parallel execution of these systems' processes releases a system $rule_i\langle\rangle$ (see (2.12)) and sends the signal syn_{Next} if rule i is executable, otherwise it sends the signal syn_{Fail}. If i results executable, then we have the following cases:

- $1 \leq j < h$: the $next_j$ process is activated. It releases a membrane that first causes the expulsion of the remainder of system Lv_j and then activates Lv_{j+1} by sending the signal syn_{j+1};
- $j = h$: the $reset_{h+1}$ process is activated. It first sends h $syn_{Execute}$ signal causing the execution of the h *rule* processes that represent the maximal set of rules to be executed, then it waits for their completion and finally it releases a membrane that, as before, forces the budding of Lv_j and then activates Lv_1 by sending the signal syn_1. The signal syn_1 concretely starts a new step of computation.

If instead i results not executable, we have two conditions:

A i is the r-th rule we find not executable within Lv_j;

B i is **not** the r-th rule we find not executable within Lv_j.

and the following cases

1. $j = 1$

   A the simulation halts. We have zero rules in the maximal set and no more rule to choose;

   B the *controller* sends another signal syn_{Start} and a new rule is evaluated;

2. $1 < j \leq h$

A the maximal set of rules we are constructing is complete: $j-1$ $\mathsf{syn}_{Execute}$ signal are sent and the *rule* processes are executed. After their execution, Lv_j is expelled and Lv_1 is activated starting a new step of computation;

B the *controller* sends another signal syn_{Start} and a new rule is evaluated.

$$
\begin{aligned}
Lv_1 &\triangleq \mathsf{syn}_1^{\perp}.\Big(rules \mid controller \mid next_1 \mid \mathsf{syn}_{TrashLv}^{\perp}.\mathsf{bud}_{Junk}\Big)\langle\diamond\rangle\\
Lv_j &\triangleq \mathsf{syn}_j^{\perp}.\Big(rules \mid controller.\mathsf{syn}_{Reset} \mid next_j \mid reset_j \mid \mathsf{syn}_{TrashLv}^{\perp}.\\
&\qquad .\mathsf{bud}_{Junk}\Big)\langle\diamond\rangle\\
Lv_h &\triangleq \mathsf{syn}_h^{\perp}.\Big(rules \mid controller.\mathsf{syn}_{Reset} \mid \mathsf{syn}_{Next}^{\perp}.reset_{h+1} \mid reset_h\\
&\qquad \mid \mathsf{syn}_{TrashLv}^{\perp}.\mathsf{bud}_{Junk}\Big)\langle\diamond\rangle\\
controller &\triangleq r\,(\mathsf{syn}_{Start}.\mathsf{syn}_{Fail}^{\perp})\\
skin &\triangleq !\,\mathsf{bud}_{Junk}^{\perp}(0).\mathsf{syn}_{TrashDone}\\
rules &\triangleq \prod_{i=1}^{r}(\mathsf{syn}_{Start}^{\perp}.\mathsf{drip}(skip_i).\mathsf{drip}(token).\mathsf{drip}(run_i))\\
next_j &\triangleq \mathsf{syn}_{Next}^{\perp}.\mathsf{drip}\Big(\mathsf{syn}_{TrashLv}.\mathsf{syn}_{TrashDone}^{\perp}.\mathsf{syn}_{j+1}\Big)\\
reset_j &\triangleq \mathsf{syn}_{Reset}^{\perp}.(j-1)\,\mathsf{syn}_{Execute}.(j-1)\,\mathsf{syn}_{Executed}^{\perp}.\mathsf{drip}(\mathsf{syn}_{TrashLv}.\\
&\qquad .\mathsf{syn}_{TrashDone}^{\perp}.\mathsf{syn}_1)
\end{aligned}
\tag{2.15}
$$

The final MDB system, in its starting configuration, that encodes the Simple SA Π is defined as follows:

$$
Encoding \triangleq skin\Big\langle !Lv_1 \circ \cdots \circ !Lv_h \circ \mathsf{syn}_1\,\langle\rangle \circ \dot{W}_1 \circ \cdots \circ \dot{W}_{k+1} \circ \dot{W}_E \circ O_1 \circ \cdots \circ O_k\Big\rangle
$$

where O_j stands for $\Big(\mathsf{mate}_{o,j}\,\langle\rangle\Big)^{n_j}$, and, given multiset $W_j = \alpha_1,\ldots,\alpha_m$, $\dot{W}_j$ stands for $\prod_i^m \Big(\mathsf{mate}_{\alpha_i,j}\,\langle\rangle\Big)$. The system $\mathsf{syn}_1\,\langle\rangle$ is the one that starts the computation by activating one of the uncountable many Lv_1 systems.

2.4 Example: The Sodium Potassium Pump

In order to explain how the translation work, we consider the modeling of the so-called sodium–potassium pump, a process which allows the exchange of sodium and potassium ions through the cell membrane. Three sodium ions are sent, through a membrane channel, outside the cell, by using one molecule of ATP. The channel is phosporilated and it changes its conformation to be open to the outside. When two potassium ions enter the channel, they are brought inside the cell, the channel is deposhorilated, and the process is ready for a new iteration. Such a process can be modeled by a Simple SA system in the following way (we stress the fact that many details are ignored here):

$$\Pi = (V, \mu, W_1, W_2, W_E, R_1, R_2)$$

where:

- $V = \{Na, K, ATP, C\}$
- $\mu = [_2[_1]_1]_2$
- $W_2 = \{B, Na^i | i \geq 1\}$
- $W_1 = \{ATP^h | h \geq 1\}$
- $W_E = \{C, K^l | l \geq 1\}$
- $R_2 = \{(Na^3, out; C, in)\} \cup \{(C, ATP, out; K^2, in)\}$
- $R_1 = \{(B, in)\} \cup \{(ATP, B, out)\}$

The system works in the following way.

We start with a certain number i of sodium ions and a single symbol B in region 2, with h molecules of energy ATP in region 1, with l copies of potassium ions in the environment, and a unique symbol C in the environment, which is used to control the whole process (one can think it encodes the state of the channel, defining in which direction it is closed).

Initially, three ions Na from region 2 are sent to the environment, while the symbol C is brought in region 2, using the rule $(Na^3, out; C, in)$. The channel is now open in the direction of the environment, but it cannot bring the potassium ions inside the cell until an energy unit is available. The symbol B in region 2 is thus sent to region 1 by the rule (B, in); here the rule (ATP, B, out) can be applied, and an energy molecule is made available in region 2.

Using the ATP molecule in region 2, we can now apply the rule $(C, ATP, out; K^2, in)$, which brings two ions K inside the cell. The energy unit ATP reaches the environment, as well as the symbol C. This last one can thus be used again to start a new cycle of the process.

We want to stress that the amount of energy in the cell is encoded in the number of symbols ATP in region 1. Each time the sodium–potassium pump is activated, one unit of such energy is consumed (a symbol ATP reaches the environment). When such energy units are over, the process cannot evolve anymore. Of course, the same is true if less than three Na ions within the cell are available to be sent in the environment, or less than two K ions are available in the environment to be brought within the cell.

2.4.1 Encoding the Simple SA Sodium Potassium pump

For lack of space we do not completely detail the encoding but we present significant parts of it and describe the *runtime* scenario of the simulation. We remember that Simple SA works with finite resources so we fix them in order to deal with the simplest case: $i = 3, h = 1, l = 2$. This means we have $|H \models h = 8$ ($i + h + l$ plus a single C and a single B). With this conditions the model should be able to complete a single cycle of the sodium–potassium pump.

For the parts shown we straightforwardly apply the encoding proposed.

First we need to define our new alphabet $\dot{V} = \{N, K, A, C, B\} \times \{1, 2, 3\}$ where index 3 corresponds to the environment region. We mapped $Na \to N$ and $ATP \to A$ for less typing. Rules also are indexed in a one to one manner:

- $(B, in) \mapsto 1$
- $(ATP, B, out) \mapsto 2$
- $(Na^3, out; C, in) \mapsto 3$
- $(C, ATP, out; K^2, in) \mapsto 4$

We suppose set $\dot{H}$ already contains the correct amount of complementary objects (see 2.3.2) for each symbol in $\dot{V}$.

Coming to the concrete encoding we detail process *run* for rule 4:

$$\begin{aligned} run_4 \triangleq\ & \overbrace{\mathsf{mate}^{\perp}_{C,2} \,.\, \mathsf{mate}^{\perp}_{A,2} \,.\, \mathsf{mate}^{\perp}_{K,3} \,.\, \mathsf{mate}^{\perp}_{K,3}}^{\longleftarrow\ \text{“testing” (consuming) objects availability}\ \longrightarrow} . \mathsf{syn}_{A} \,.\, \mathsf{mate}^{\perp}_{Token} \,.\, \mathsf{syn}^{\perp}_{A} . \textit{undo-skip}_4 . \mathsf{syn}_{TrashSkip} \,. \\ & .\, \mathsf{syn}^{\perp}_{TrashDone} . \textit{inc-}\overline{\gamma_i} .\, \mathsf{drip}(rule_4) . \mathsf{drip}\big(\mathsf{syn}_{TrashRun} \,.\, \mathsf{syn}^{\perp}_{TrashDone} \,.\, \mathsf{syn}_{Next}\big) \mid \mathsf{syn}^{\perp}_{TrashRun} \,.\, \mathsf{bud}_{Junk} \\ rule_4 \triangleq\ & \mathsf{syn}^{\perp}_{Execute} \,. \overbrace{\mathsf{drip}(\mathsf{mate}_{C,3}) .\, \mathsf{drip}(\mathsf{mate}_{A,3}) .\, \mathsf{drip}(\mathsf{mate}_{K,2}) .\, \mathsf{drip}(\mathsf{mate}_{K,2})}^{\longleftarrow\ \text{“moving” (creating) objects}\ \longrightarrow} . \\ & . \textit{dec-}\overline{\omega_i} . \mathsf{syn}_{Executed} \end{aligned} \tag{2.16}$$

Even if our encoding actually “consumes” and “creates” objects, we faithfully simulate the *purely communication based* computation of Simple SA: the created objects corresponds to the consumed one with a new position in the membrane structure.

Processes *inc*-$\overline{\gamma_i}$ and *dec*-$\overline{\omega_i}$ take care of the complementary objects in the simulation and are fully described in section 2.3.3. Here we just remind that for (2.8) every time we consume an object a we must create an instance of the corresponding complementary object $\overline{a}$, and vice versa, for every created object b we must consume an instance of $\overline{b}$.

We define τ as follow

$$\begin{aligned} \tau \triangleq\ & \mathsf{syn}_B \,.\, \mathsf{mate}^{\perp}_{Token} . \textit{undo-skip}_4 . \textit{undo-run}_4 .\, \mathsf{syn}_{TrashRun} \,.\, \mathsf{syn}^{\perp}_{TrashDone} \,. \\ & .\, \mathsf{drip}\left(\mathsf{syn}_{TrashSkip} \,.\, \mathsf{syn}^{\perp}_{TrashDone} \,.\, \mathsf{syn}_{Fail}\right) \end{aligned}$$

and present the encoding of $skip_4$

$$\begin{aligned} skip_4 \triangleq\ & \overbrace{8\, \mathsf{mate}^{\perp}_{\overline{\mathsf{mate}_{A,2}}} . \tau \mid 8\, \mathsf{mate}^{\perp}_{\overline{\mathsf{mate}_{C,2}}} . \tau \mid 7\, \mathsf{mate}^{\perp}_{\overline{\mathsf{mate}_{K,3}}} . \tau}^{\longleftarrow\ \text{testing unavailability (in parallel!)}\ \longrightarrow} \mid 3\, \mathsf{syn}^{\perp}_{B} \,.\, \mathsf{syn}_{UndoDone} \\ & \mid\ \mathsf{syn}^{\perp}_{TrashSkip} \,.\, \mathsf{bud}_{Junk} \end{aligned} \tag{2.17}$$

Since $rule_4$ requires 3 distinct objects we have $t_i = 3$ in process $skip_4$ and need to test only these resources. Having a total (fixed) number of eight symbols[7] in our system if, for example, we successfully complete sequence $8\, \mathsf{mate}^{\perp}_{\overline{\mathsf{mate}_{A,2}}}$, for the complementary objects logic this would mean that there are at least 8 symbols that differ from symbol $\mathsf{mate}_{A,2}$ and so, that there is no symbol $\mathsf{mate}_{A,2}$ available. The encoding of process *run* and *skip*

[7]The complementary symbols are not counted since they are just *meta*–symbols.

for the most complex rule in our simulation should let you easily guess the encoding of the remaining rules.

So let's suppose our simulation starts. Since we are simulating the parallel semantics of Simple SA we must determine a maximal set of rules and simulate their parallel execution. As stated in 2.3.4 we realize this by means of systems Lv_i. System $\mathsf{syn}_1\,\langle\rangle$, which is provided in the initial configuration, activates a system Lv_1. The system Lv_1 must determine the first executable rule and *add* it to the maximal set in construction. So far we have that the only *active* process is *controller* on the activated Lv_1 system. The total number of rules in our model is four so

$$controller \triangleq 4\,(\mathsf{syn}_{Start}\,.\,\mathsf{syn}^{\perp}_{Fail})$$

To us it is immediate that the only fireable rules are 3 and 1, but our model can only randomly choose a rule and then test if it is executable or not. In the worst case process *controller* will first trigger the tests for rules 2 and 4 before choosing either rule 3 or rule 1 as executable.

$$rules \triangleq \mathsf{syn}^{\perp}_{Start}\,.\,\mathsf{drip}(skip_1).\,\mathsf{drip}(token).\,\mathsf{drip}(run_1) \mid \mathsf{syn}^{\perp}_{Start}\,.\,\mathsf{drip}(skip_2).\,\mathsf{drip}(token).\,\mathsf{drip}(run_2) \mid$$
$$\mid \mathsf{syn}^{\perp}_{Start}\,.\,\mathsf{drip}(skip_3).\,\mathsf{drip}(token).\,\mathsf{drip}(run_3) \mid \mathsf{syn}^{\perp}_{Start}\,.\,\mathsf{drip}(skip_4).\,\mathsf{drip}(token).\,\mathsf{drip}(run_4) \tag{2.18}$$

The *controller* process sends a syn_{Start} signal and waits for a signal syn_{Fail}. All the four processes in (2.18) are competing for the signal syn_{Start} sent by the *controller* but only one can receive it. Let' suppose the signal is received by the process that tests rule number 1.

The testing of rule 1 starts by releasing the systems $skip_1\,\langle\rangle$, $run_1\,\langle\rangle$ and $token\,\langle\rangle$. Since we have an object B in region 2, rule 1 is executable: process run_1 releases the system $rule_1\,\langle\rangle$. Please note that process $rule_1$ will execute only after receiving a signal $\mathsf{syn}_{Execute}$. The remainder of system $run_1\,\langle\rangle$ is expelled out to the environment and the signal syn_{Next} is produced.

So far system Lv_1 has accomplished its goal by adding rule 1 to the maximal set we are constructing. The signal syn_{Next} is received by the process $next_1$ on system Lv_1, the remainder of system Lv_1 is expelled to the environment and the signal syn_2 is produced. This activates a system Lv_2 which goal is to find another executable rule. We want to stress out that we have not yet executed process $rule_1$, it is still awaiting for a signal $\mathsf{syn}_{Execute}$, but we have indeed *locked* resources for its execution. Clearly the *locked* resources will not be available to system Lv_2.

System Lv_2 undergoes, more or less, the same sequence we just described for system Lv_1. Briefly, a new system $rule_3\,\langle\rangle$ is released, the remainder of system Lv_2 is discarded to the environment and a signal syn_3 is produced.

It should be easy too see no more rules are executable. System Lv_3 will indeed test all the four rules of our simulation but no run_i process is going to succeed. Every test will indeed terminate by sending a syn_{Fail} signal. Since four syn_{Fail} signals are produced the *controller* process will complete its sequence and so the signal syn_{Reset} will be produced (see (2.15)). The signal produced triggers the $reset_3$ process on system Lv_3.

So far our simulation have completed the maximal set of rules (with rules 1 and 3), and now we must execute the chosen rules and start a new step of computation.

It is the $reset_3$ process that triggers the execution of processes $rule_1$ and $rule_3$ by producing two $\mathsf{syn}_{Execute}$ signals. After the execution, processes $rule_1$ and $rule_3$ reply with one $\mathsf{syn}_{Executed}$ signal each. The $reset_3$ process receives the two signals and forces the expulsion to the environment of the remainder of system Lv_3. Finally, it produces a new syn_1 signal. As you can easily see the produced signal triggers a new Lv_1 system and a new step of computation starts.

At this point we have successfully simulated the parallel execution of rules 1 and 3 by simulating the transport of a B symbol from region 2 to region 1, the transport of three N symbols from region 2 to the environment and the transport of a single C symbol from the environment to region 2. To simulate this we deleted the symbols

$$\mathsf{mate}_{B,2}, 3\,\mathsf{mate}_{N,2}, \mathsf{mate}_{C,3}$$

and produced the following

$$\mathsf{mate}_{B,1}, 3\,\mathsf{mate}_{N,3}, \mathsf{mate}_{C,2}$$

We chose some “lucky” branch in our simulation in order to avoid the description of a long and tedious *skip* process, but you should note that, given the starting configuration, in whichever order we test our rules we obtain a maximal set containing exactly rules 3 an 1 since it is the unique maximal set possible.

The simulation undergoes two more computation step executing rule 2 in the first step and rule 4 in parallel with rule 1 in the second one. After completing this last step the simulation will start a new step searching for a new executable rule by means of a system Lv_1. Since we can not execute any rules four syn_{Fail} signals are produced, the *controller* process on Lv_1 completes and, with no active process, the entire simulation stops (see definition of Lv_1 in 2.15).

A single cycle of the sodium–potassium pump has been simulated.

2.5 Conclusion and final remarks

This chapter is a contribution to the recently started investigations aiming to bridge two research areas which are both inspired by the functioning of the living cell: Membrane computing and Brane calculi.

We have presented a simulation of a variant of Membrane systems by means of Brane calculi based on the three operations Mate/Bud/Drip. As the main idea behind the chapter was to investigate the use of these two frameworks in Systems Biology area, we chose to start from simple formalisms: we chose MBD Brane calculus, which is non-Turing equivalent, to simulate a variant of P systems called Simple Symport/Antiport P systems (Simple SA); the computations of this last type of systems is, in fact, purely based on communications of objects and it is obtained by coupling chemicals and by changing their position with respect to the compartments defined by the membrane structure used, and not by creating or destroying some of them. Their behaviour is thus closely related to the biological trans-membrane communication, and they can be effectively used in describing such processes. Moreover, compared with basic Symport/Antiport P systems, Simple SA is a less powerful computational device, but its limited resources scenario makes it a more realistic model for Systems Biology.

We recall here that, as pointed out in the previous section, the encoding we have proposed is non deterministic: if the Simple SA system to be simulated is non deterministic, then also the MDB encoding is non deterministic. It remains an open problem to show if it is possible or not to simulate Simple SA systems using deterministic MBD Brane calculus.

Many other related research topics remains to be investigated: the simulation of other classes of P systems by means of MBD Brane calculus, the use of Brane calculus using the Pino/Exo/Phago operations, and the use of different types of parallel semantics within the framework of Brane calculi are the directions we plan to explore in the near future.

Chapter 3

κ and nanoκ: the *local-implementation* problem

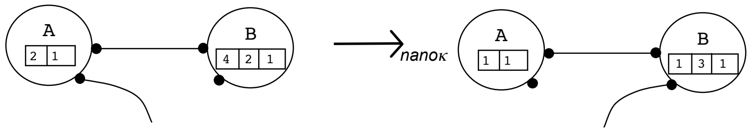
In this chapter we address the *local-implementation* problem, already introduced in Chapter 1, between κ calculus and nanoκ calculus. More precisely we study the implementation of κ into nanoκ. As introduced in Chapter 1 the former is a model for molecular biology that rewrites graphs of molecules in one step; the latter is a calculus similar to κ that only admits binary interactions. We give a solution of the local-implementation of κ in nanoκ that is divergent and we show the nonexistence of deterministic solutions retaining "reasonable" properties.

3.1 Introduction

As already detailed the κ calculus has been introduced for modeling molecular biology in a formal way. It is a graph rewriting system where nodes represent molecules and edges represent bonds. Nodes retain a finite information, typically about the shape of the molecule or about connected molecules. The semantics allows monotone rewritings of finite graphs whose nodes are in specific states into finite graphs in such a way that changes to a solution are always *localized* to the rewriting part. Monotonicity constraints rewritings to either create or destruct molecules and bonds.

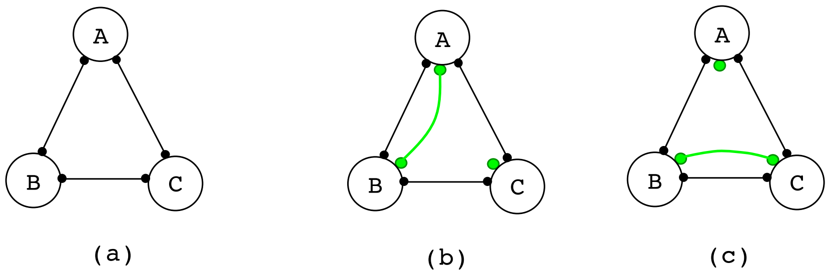
The κ calculus, being as much simple and close to biology as possible, admits rewriting rules where several molecules may interact at a time. The question that was raised already in [22] is whether κ calculus may be implemented in a calculus with binary reactants only or not. This problem, called *local-implementation*, had a positive answer in a variant of κ calculus – the mκ calculus – with binary rewriting rules and *multiedges*. The idea was to use these multiedges as logs of the reactions. The check that reactants are connected, as prescribed in the left-hand side of the reaction, reduced to verify that the connected molecules all share the same log.

Some years later, a new formalism – the nanoκ calculus – similar to κ and mκ, was introduced for modelling nano-technologies [17]. This calculus has binary interactions (as mκ) but no multiedge is admitted (as in κ). Some expressive power is recovered by admitting a new binary rule – the *exchange* – that allows an end of a bond to be passed from one molecule to another. This rule, which is illustrated in the following picture

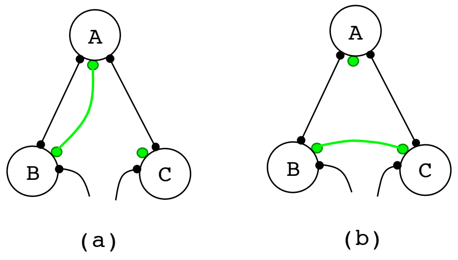


has been motivated by process calculi, where it is customary to receive a channel name and communicating on such a channel in the continuation. However, it was not clear whether nanoκ calculus was expressive enough for implementing the κ calculus.

In this chapter we demonstrate that the local-implementation of κ into nanoκ is possible and we define the protocol. The solution is similar to the one from κ to mκ, except that the check verifying the proper connection of reactants is performed by percolating a bond among them. To illustrate the point, consider the structure (a) below, and assume that *A* wants to verify that the *B* and *C* to which it is connected are connected between them.



Molecules *A* and *B* create a bond (the one to be exchanged), as illustrated in picture (b). Then the connection between *A* and *C* makes this bond to be exchanged as in figure (c). This is a successful state because *B* and *C* are connected by two bonds (and the success is eventually reported to A). It is worth to notice that this algorithm fails if B and C are not connected, as illustrated below:



(there is only one bond connecting B and C).

The local-implementation protocols proposed in [22, 18] and in this chapter avoid deadlocks by admitting divergent computations. A relevant question is whether a deterministic, not-divergent local-implementation encoding κ into a calculus with binary reactants, such as mκ or nanoκ, does exist or not. In this chapter we show that, under "reasonable" constraints, such a protocol does not exist. This case is close to the comparison of the expressive power of synchronous and asynchronous π calculi done by Palamidessi [45]. As in that case, we require the local-implementation protocol to be uniform – homomorphic with respect to parallel composition and preserving the connectedness of molecules – and semantically reasonable – preserving termination and the complexes.

However, it turns out that these constraints are inadequate to exclude convergent local-implementation protocols of κ because these protocols must also redefine the reaction rules. That is, since every κ reaction is encoded by a set of low-level ones, we need to regulate this set in order to avoid misbehaviours. For example, a malicious protocol (which is uniform and semantically reasonable) might grab more material then necessary for the reaction (in the worst case, all the material in the solution) and release it after the reaction has been performed, thus being inconsistent with the locality principle of the κ family.

In order to localize the effects of protocols we also add a constraint called *twinning*: some reactions between those implementing a κ reaction $\mathsf{L} \rightarrow \mathsf{R}$ have a twin one in the protocol that undoes the effects on bonds. In particular, if $\mathsf{L} \rightarrow \mathsf{R}$ is a creation then every destruction in its protocol has a corresponding creation restoring the bonds in the reactants. This means that if the protocol of a creation unbinds a molecule then the released molecule may be rebound, thus yielding either a previous solution – and the computation may diverge – or a previous complex (with a different state). In this latter case, if the complex is too big with respect to the complexes in R then the protocol is not semantically reasonable.

The chapter is structured as follows. In Section 3.2 we define the calculi of the κ family (κ, mκ, and nanoκ calculus). In Section 3.3 we discuss the local-implementation problem for κ, recall the protocol in mκ and define the protocol in nanoκ calculus. The Section 3.4 discusses the problem of not-divergent local-implementation protocols.

3.2 The κ family: syntax and semantics

Two disjoint countable sets of names will be used: a set of *species*, ranged over by A, B, C, $\cdots$; and a totally ordered set of *bonds*, ranged over by x, y, z, $\cdots$. Species are sorted according to the number of *fields* and *sites* they possess. Let $\mathfrak{s}_f(\cdot)$ and $\mathfrak{s}_s(\cdot)$ be two functions returning naturals; the numbers 1, 2, $\cdots$, $\mathfrak{s}_f(A)$ and 1, 2, $\cdots$ $\mathfrak{s}_s(A)$ are respectively the fields and the sites of A. ($\mathfrak{s}_f(A) = 0$ means there is no field; $\mathfrak{s}_s(A) = 0$ means there is no site). In the following, fields are ranged over by h, i, j, $\cdots$; sites are ranged over by a, b, c, $\cdots$.

Sites may be either *bound* to other sites or *unbound*, i.e. not connected to other sites. The state of sites are defined by maps, called *interfaces* and ranged over by σ, ρ, $\cdots$. Given a species A, its interfaces are *partial functions* from $\{1, \cdots, \mathfrak{s}_s(A)\}$ to the set of bonds or a special empty value ε. A site a is bound with bond x in σ if $\sigma(a) = x$; it is unbound if $\sigma(a) = \varepsilon$. For instance, if A is a species with three sites, $(2 \mapsto x, 3 \mapsto \varepsilon)$ is one of its interfaces. In order to ease the reading, we write this map as $2^x + 3$ (the empty value is always omitted). This interface σ does not define the state of the site 1, which may be bound or not. In the following, when we write $\sigma + \sigma'$ we assume that the domains of σ and σ' are disjoint.

Fields represent the internal state of a species. The values of fields are defined by maps, called *evaluations*, and ranged over by u, v, $\cdots$. For instance, if A is a species with three fields, $[1 \mapsto 5, 2 \mapsto 0, 3 \mapsto 4]$ is a possible evaluation. As before, we write this map as $1^5 + 2^0 + 3^4$. We assume there are finitely many internal states, that is every field h is mapped into a *finite set of values*. As for interfaces, we use *partial* evaluations and, when we write the union of evaluations $u + v$, we implicitly assume that the domains of u and v are disjoint.

Definition 3.1 *A* molecule $A[u](\sigma)$ *is a term where* u *and* σ *are respectively a total evaluation of* A *and a total interface of* A. Solutions*, ranged over by* S, T, $\cdots$*, are defined by the following grammar*

$$\mathsf{S} \quad ::= \quad A[u](\sigma) \ \mid \ \mathsf{S}\,\texttt{,}\,\mathsf{S}$$

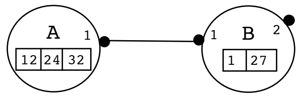
The operator "`,`*" is assumed to be associative, so* $(\mathsf{S}\,\texttt{,}\,\mathsf{S}')\,\texttt{,}\,\mathsf{S}''$ *is equal to* $\mathsf{S}\,\texttt{,}\,(\mathsf{S}'\,\texttt{,}\,\mathsf{S}'')$ *(and we always omit parentheses).*

Solutions retain the property that bonds always occur at most twice. A solution is proper *if bonds occur exactly twice. Let* $\mathtt{bonds}(\mathsf{S})$ *be the bonds of* S*.*

The calculi in the κ family retain the same terms and differ for the shape of reactions. We define the reactions and the transition system of the κ calculus and, later on, we discuss the reactions of the other calculi in the family. Few preliminary definitions are in order:

- we write $\sigma \leq \sigma'$ if $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ and, for every i, if $\sigma(i) \neq \varepsilon$ then $\sigma(i) = \sigma'(i)$ (the two interfaces may differ on sites mapped to the empty value ε by σ: σ' may map such sites to bonds);
- a *pre-solution* is a sequence of terms $A[u](\sigma)$ where u and σ are partial functions and bonds occur at most twice;
- a pre-solution is *proper* if it retains the property that bonds occur exactly twice.

The κ calculus retains an intelligible graphical notation [21]. For example the solution $A[1^{12} + 2^{24} + 3^{32}](1^x)\,\texttt{,}\,B[1^1 + 2^{27}](1^x + 2)$ is represented by the picture



The formal translation from solutions to graphs is given below.

Definition 3.2 *Let* graph($\cdot$) *be a function from solutions to graphs where nodes have sites and an internal state:*

1. graph($A[u](\sigma)$) *is the graph with a single node labeled A, sites in* $\{1, \cdots, \mathfrak{s}_s(A)\}$*, and a tuple of values. The site i is labeled with* $\sigma(i)$ *(i.e. the bond, if any); the j-th element of the tuple has value* $u(j)$ *(i.e. the j-th field value);*
2. graph(S, S′) *is the union graph of* graph(S) *and* graph(S′) *where sites labeled with the same name are connected by an edge, and their common name is erased.*

graph(S) *is called the* underlying graph *of* S.

Two molecules in a solution S *are* connected *if there is a path of bonds in* graph(S) *that connects the corresponding nodes.*

The definitions of underlying graph and connectedness easily extend to pre-solutions by taking the fields and the sites that are specified. Connectedness allows us to define complexes: a complex is a bunch of connected molecules where bonds occur exactly twice. We will extensively use the graphical notation in the rest of the chapter – indeed, it has been already used in the Introduction – sometimes replacing fields with colors. In particular, we will use graphs for describing reactions – see below.

Definition 3.3 *Reactions of* κ *calculus are either* creations C*, or* destructions D. *The format of creations is*

$$\begin{aligned} A_1[u_1](\sigma_1), \cdots, A_n[u_n](\sigma_n) \to \quad & A_1[u'_1](\sigma'_1), \cdots, A_n[u'_n](\sigma'_n), \\ & B_1[v_1](\phi_1), \cdots, B_k[v_k](\phi_k) \end{aligned}$$

where, for every i, $\mathrm{dom}(u_i) = \mathrm{dom}(u'_i)$ *and* $\sigma_i \leq \sigma'_i$*, reactants and products are proper pre-solutions, the products are connected, and* v_i *and* ϕ_i *are total. The format of destructions is*

$$A_1[u_1](\sigma_1), \cdots, A_n[u_n](\sigma_n) \to A_{i_1}[u'_{i_1}](\sigma'_{i_1}), \cdots, A_{i_k}[u'_{i_k}](\sigma'_{i_k})$$

where, $i_1, \cdots, i_k$ *is an ordered sequence in* $[1..n]$*, for every* i_j*,* $\mathrm{dom}(u_{i_j}) = \mathrm{dom}(u'_{i_j})$ *and* $\sigma_{i_j} \geq \sigma'_{i_j}$*, reactants and products are proper pre-solutions, the reactants are connected, and, for every* $j \notin \{i_1, \cdots, i_k\}$*,* u_j*,* σ_j *are total.*

Creations produce new bonds between two unbound sites and/or synthesize new molecules (that must be connected to the molecules in the left-hand side). Destructions behave in the other way around [1]. We assume that reactants and products always have at least one term.

Example 3.1 We illustrate few κ calculus reactions that corresponds to biochemical reactions. We only discuss creations.

1. The hydrogen gas is the combination of two hydrogen atoms:

$$H[1], H[1] \quad \rightarrow \quad H[1^x], H[1^x]$$

2. The homeotrimerization is a combination of three monomers of the same species:

$$A[1+2], A[1+2], A[1+2] \quad \rightarrow \quad A[1^x + 2^y], A[1^y + 2^z], A[1^z + 2^x]$$

3. As an example of synthesis, we consider Escherichia Coli that has to synthesize galactosidase (Gal) and permease (Per) when the repressor is absent (field rep of RNAp equal to 0):

$$\begin{aligned} RNAp[rep^0](s_{Gal} + s_{Per}) \rightarrow \quad & RNAp[rep^1](s^y_{Gal} + s^z_{Per}), \\ & Gal[loaded^0](lac + s^y), Per(lac + s^z) \end{aligned}$$

(With an abuse of notation, here and below, identifiers are used instead of numbers for addressing fields and sites.)

□

The operational semantics of κ calculus is a *reduction semantics*, which requires a couple of standard definitions.

- The structural equivalence between solutions, noted $\equiv$, is the least equivalence satisfying the following two rules (we remind that solutions are already quotiented by associativity of " , "):

[1]The terms creation and destruction have been preferred to *complexation* and *decomplexation* used in [22, 33] because they have a more neutral chemical meaning.

1. $\mathsf{S},\mathsf{T} \equiv \mathsf{T},\mathsf{S}$;
2. $\mathsf{S} \equiv \mathsf{T}$ if there exists an injective renaming $\imath$ on bonds such that $\mathsf{S} = \imath(\mathsf{T})$.

- Let $\mathsf{S} = A_1[u_1](\sigma_1), \cdots, A_n[u_n](\sigma_n)$ be a pre-solution. We say that $\mathsf{T} = A_1[u_1 + u'_1](\sigma_1 \circ \imath + \sigma'_1), \cdots, A_n[u_n + u'_n](\sigma_n \circ \imath + \sigma'_n)$ is an $(\imath, u'_1, \sigma'_1, \cdots, u'_n, \sigma'_n)$-*instance* of S if $\imath$ is an injective renaming on bonds and the maps $u_i + u'_i$ and $\sigma_i \circ \imath + \sigma'_i$ are *total* with respect to the species A_i.

Using structural equivalence it is possible to identify solutions that should not be kept distinct, such as $H(b^x), H(b^x), H(b^z), H(b^z) \equiv H(b^y), H(b^k), H(b^k), H(b^y)$. We also notice that an instance may not be necessarily a proper solution. For example $A[u^0](1^y + 2^x)$ is an $([x \mapsto y], [u \mapsto 0], [2 \mapsto x])$-instance of $A(1^x)$, but it is not a proper solution (bonds occur once).

Definition 3.4 *The reduction relation of the κ calculus, written $\longrightarrow$, is the least relation satisfying the rules:*

1. *let* $\mathsf{P} \longrightarrow \mathsf{P}', \mathsf{Q}$ *be a creation and* S *is an* $(\imath, u'_1, \sigma'_1, \cdots, u'_n, \sigma'_n)$*-instance of* P*,* S' *is an* $(\imath, u'_1, \sigma'_1, \cdots, u'_n, \sigma'_n)$*-instance of* P' *and* T *is an* $(\imath, \emptyset, \emptyset, \cdots, \emptyset, \emptyset)$*-instance of* Q*. Then* $\mathsf{S} \longrightarrow \mathsf{S}', \mathsf{T}$*;*
2. *let* $\mathsf{P} \longrightarrow \mathsf{Q}$ *be a destruction and* S *is an* $(\imath, u'_1, \sigma'_1, \cdots, u'_n, \sigma'_n)$*-instance of* P *and* T *is an* $(\imath, u'_{i_1}, \sigma'_{i_1}, \cdots, u'_{i_k}, \sigma'_{i_k})$*-instance of* Q*. Then* $\mathsf{S} \longrightarrow \mathsf{T}$*;*
3. *let* $\mathsf{S} \longrightarrow \mathsf{S}'$ *and* $(\mathsf{bonds}(\mathsf{S}') \setminus \mathsf{bonds}(\mathsf{S})) \cap \mathsf{bonds}(\mathsf{T}) = \emptyset$*; then* $\mathsf{S}, \mathsf{T} \longrightarrow \mathsf{S}', \mathsf{T}$*;*
4. *let* $\mathsf{S} \equiv \mathsf{S}'$*,* $\mathsf{S}' \longrightarrow \mathsf{T}'$*, and* $\mathsf{T}' \equiv \mathsf{T}$*; then* $\mathsf{S} \longrightarrow \mathsf{T}$*.*

The definition of reduction regards reactions as *schemas*. Namely, a reaction such as $Na[e^0](1), Cl[e^0]1 \rightarrow Na[e^1](1^x), Cl[e^{-1}](1^x)$ only addresses the fields and the the sites of the reactants that are useful for the reaction. For example, it may be the case that Na retains a site to be used for other complexes (the *sodium peroxide*, for example). In this case, the rule may be applied either to $Na[e^0](1 + 2)$, where the site is unbound, or to $Na[e^0](1 + 2^z)$. In this latter case, the reaction is instantiated as the reduction:

$$Na[e^0](1 + 2^z), Cl[e^0](1) \longrightarrow Na[e^1](1^x + 2^z), Cl[e^{-1}](1^x) \tag{3.1}$$

Items 3 and 4 of Definition 3.4 allow one to derive the reductions of bigger solutions, such as

$$Na[e^0](1 + 2^z), Cl[e^0](1), Na[e^1](1^x + 2^z), Cl[e^{-1}](1^x) \tag{3.2}$$

Reduction (3.1) may be used for deriving a reduction of the first two terms of (3.2), however it cannot be lifted to the whole solution because the bond created in (3.1) clashes with a bond already present in the solution. In this case, one derives a reduction for the structural equivalent solution $Na[e^0](1 + 2^z), Cl[e^0](1), Na[e^1](1^y + 2^z), Cl[e^{-1}](1^y)$ and then a reduction of (3.2) is got by applying the last rule of Definition 3.4. It is straightforward to verify that the check of bond-clashes and the properness of reactants and products imply that proper solutions always reduce to proper solutions.

A basic property of κ calculus (and the other calculi of the family) is *locality*: if a sub-solution reduces then the reduction may be lifted to the whole solution without any effect on the remaining part – a direct consequence of Definition 3.4. In other words, the effects of a reduction are *localized* to the parts of molecules specified in the reaction rules.

Two other calculi, similar to κ calculus, have also been studied: the mκ calculus and the nanoκ calculus [2].

Definition 3.5 *The* mκ *calculus has species and solutions as the* κ *calculus but*

1. *bonds may occur more than twice in a solution (multi-edges are admitted: these are called* group-names *in [22]);*

2. *reactants consist of at most two terms and, as well as products, may be not proper.*

The nanoκ *calculus has species and solutions as the* κ *calculus but*

[2]The following definitions of mκ calculus and nanoκ calculus are a bit different from those in [22, 17]. In particular we admit creations of several terms at once, while this was not admitted in [22] and was not considered in [17]. However, these differences are not meaningful in the rest of the chapter.

1. *reactants consist of at most two terms;*

2. *there is a third type of reactions, the* exchanges E, *whose format is*

$$A[u](\rho), B[v](\psi) \longrightarrow A[u'](\rho'), B[v'](\psi')$$

with either $\rho = \rho'$ *and* $\psi = \psi'$ *or* $\rho = a^x + c^y + \rho''$, $\rho' = a + c^y + \rho''$ *and* $\psi = b + d^y + \psi''$, $\psi' = b^x + d^y + \psi''$.

Reactions of mκ and nanoκ retain a process-calculus flavour since they amount to interactions between at most two terms. However, in order to recover (at least, to some extent) the expressiveness of κ, reactions are extended with two different features:

- in mκ one may write $A[1^0](1^x), B[1^0](2^x) \longrightarrow A[1^1](1^x), B[1^1](2^x), C[0^1 + 1^1](1^x + 2 + 3)$ meaning that C is created and complexed both with A and with B: the multi-edge x represents the skeleton of the complex;

- in nanoκ one may write $A[1^0](1^x + 2^y), B[1^0](1^y + 2) \longrightarrow A[1^1](1 + 2^y), B[1^1](1^y + 2^x)$ meaning that the edge x *migrates* from A to B: the other end of the edge remains untouched. We notice that exchanges never modify the connectedness of a solution.

3.3 The local-implementation problem

The κ calculus allows for several many molecules to react in a reaction. The *local-implementation problem* questions whether it is possible to obtain *the same behaviour* with "elementary" reactions involving at most two molecules. This problem got a positive answer when the elementary reactions were those of mκ [22, 18]. κ-reactions were decomposed in sequences of mκ reactions by using the following protocol:

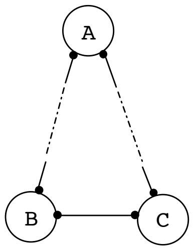
1. *Recruitment step*: every κ reaction has a unique *spanning tree* covering its reactants; in this step all the reactants are recruited (by using ad-hoc sites in the encoded molecules). At the end of the step, all the molecules in the spanning tree share a common multi-edge.

2. *Later contacts step*: the spanning tree is inadequate when the left-hand sides of κ reactions are not trees, such as $A(1^x + 2^y), B(1^x + 2^z), C(1^y + 2^z)$. In this case, letting A be the root of the spanning tree, the protocol has to verify that B and C are connected by means of the two sites 2 *and share the same multi-edge*.

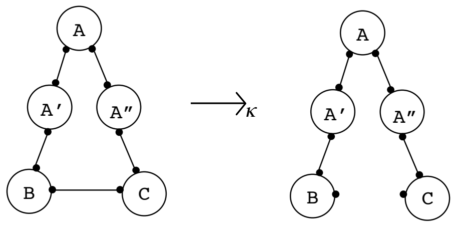
3. *Phase shift step*: when the left-hand side of the reaction has been completely checked, the (κ) reaction may occur and the product is generated. The κ reaction is implemented as a sequence of mκ reactions.

It is clear that every mκ reaction of steps (1) and (2) may fail; for this reason such reactions are *reversible* and the protocol has been proved to be correct with respect to *weak coupled simulation* in [22] and *weak bisimulation* in [18] (which are both insensitive to divergence).

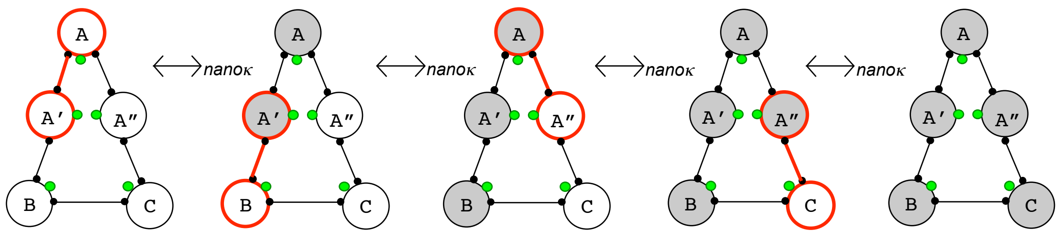
The protocol used for local-implementation κ in mκ may be adapted to nanoκ. In particular, the steps are the same as those described in [22, 18], except for the later contacts, where the spanning tree must be checked for the presence of additional bonds between the molecules therein. This case is illustrated in the following picture – called *the triangular trade* –, where A is a common parent of B and C in the spanning tree (one may take A as the unique parent at highest depth) and the bond between B and C must be verified (they have already been recruited and the protocol may fail because of the absence of such a bond).



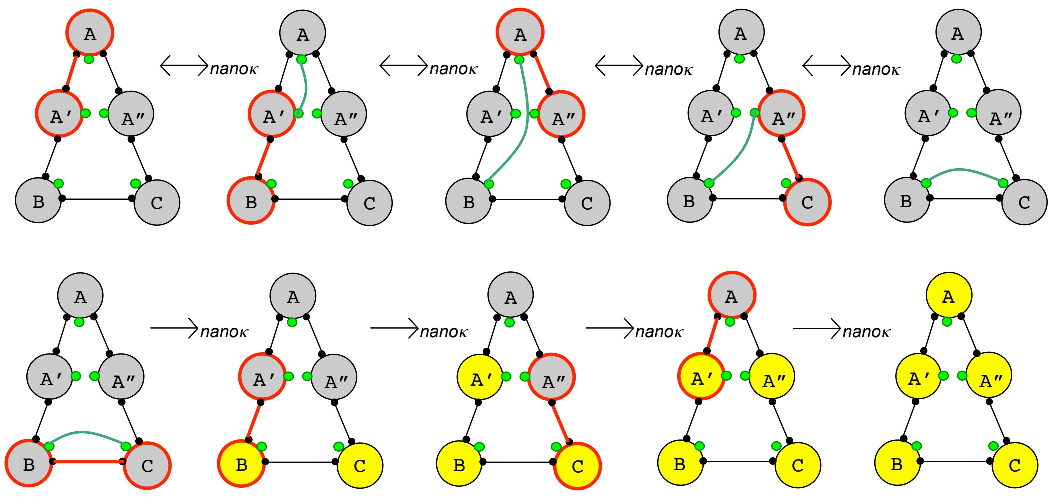
Without loss of generality, we illustrate the protocol for a κ reaction rewriting a triangular trade:



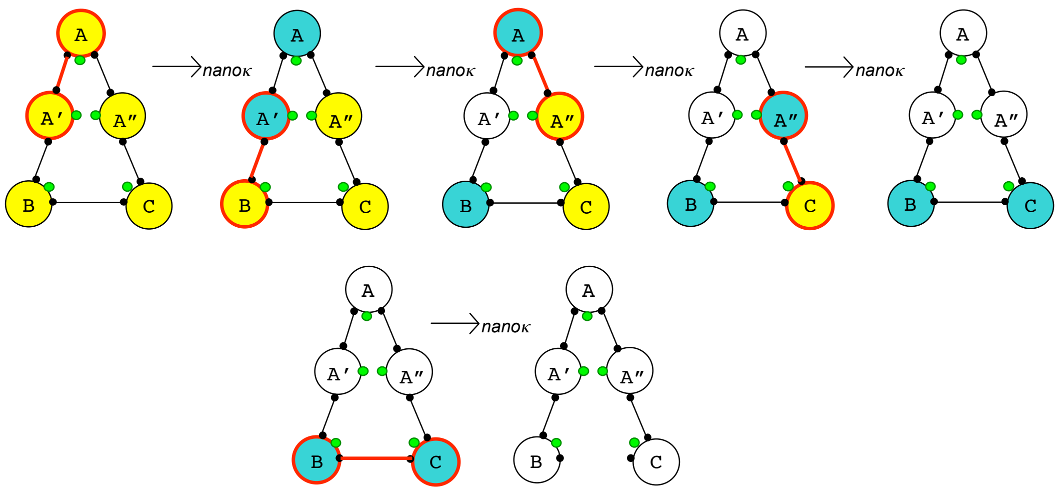
(the arrow is indexed by κ in order to avoid ambiguities). To ease the understanding, the description is pictorial. In this reaction, fields have been omitted for simplicity. Molecules are encoded into lower-level ones having an additional site (for a bond to be exchanged) and an additional field that, in the following protocol, will store the color. Colors are used to mark the step of the protocol in the molecules. The recruitment step is the following sequence of *reversible* nanoκ reactions, where the spanning tree is assumed to be the right-hand side of the κ reaction. At the end of the step, every molecule is gray.



The later contacts step checks that B and C are actually bound each other (this may be not the case, in general, because B may be bound to a C different from the recruited one).



In order to verify that B and C are bound as required, a new edge is created by the root A and it is percolated among the nodes by means of exchange reactions till reaching the configuration depicted in the rightmost complex of the first line. Up-to now, every nanoκ reaction is reversible because the protocol may fail. On the contrary, once the double connection between B and C is verified – leftmost reaction in the second line – , the protocol cannot fail anymore and the reactions are unidirectional. The success is reported to the root A of the spanning tree by coloring the molecules in yellow. (Actually, every reaction has to be reversible when there are several triangular trades in the reactants of κ.) At this stage the phase shift step may begin and the effects of the κ reaction may be implemented. This is described by the following reactions.



Following the same pattern of [22, 18], it is possible to demonstrate the correctness of the local-implementation protocol in nanoκ calculus. The formal proof is omitted.

3.4 Divergence and determinism

The local-implementation protocols proposed in [22, 18] and in the previous section are divergent: the protocols backtrack in case of failures that may happen in the recruitment or the later contacts steps. The combination of forward and backward computations produce (infinite) loops. The questionable issue is whether a deterministic, not-divergent protocol encoding κ into a calculus with binary interactions does exist or not.

We remark that, the local-implementation protocol $[\![\cdot]\!]$ must encode both a solution – the initial one – and a set of κ reactions. Following Palamidessi [45], let $[\![\cdot]\!]$ be *uniform* if

- it is homomorphic with respect to "$\texttt{,}$", namely $[\![\mathsf{S}\,\texttt{,}\,\mathsf{T}]\!] = [\![\mathsf{S}]\!]\,\texttt{,}\,[\![\mathsf{T}]\!]$;
- it is renaming preserving, namely for every injective renaming $\imath$ on bonds of S there exists an injective renaming $\jmath$ such that $[\![\imath(\mathsf{S})]\!] = \jmath([\![\mathsf{S}]\!])$.

and be *semantically reasonable* if

- it preserves the relevant observables and the termination properties.

Uniformity guarantees that the degree of distribution of the solution is maintained by the encoding, i.e. no coordinator is added, and that the encoding does not depend on bonds. It is worth to notice that, in our case, $[\![\cdot]\!]$ might introduce new fields and new sites in the nanoκ molecules (called low-level fields and sites in the following). In addition, $[\![\cdot]\!]$ must redefine κ reactions in order to fit with the new schemas of nanoκ. We therefore extend Palamidessi's notion of uniformity of $[\![\cdot]\!]$ with the following requirements:

- for every κ reaction $\mathsf{L} \to \mathsf{R}$, $[\![\mathsf{L} \to \mathsf{R}]\!] = \{\mathsf{L}_1 \to \mathsf{R}_1, \cdots, \mathsf{L}_m \to \mathsf{R}_m\}$, where $\mathsf{L}_i \to \mathsf{R}_i$ are nanoκ reactions;
- (this is for simplicity) $[\![A[u](\sigma)]\!] = A[[\![u]\!] + v]([\![\sigma]\!] + \rho)$, that is $[\![\cdot]\!]$ preserves the granularity but may augment fields and sites. ($[\![u]\!]$ and $[\![\sigma]\!]$ may also have larger domains than u and σ, respectively.)

As regards the semantics reasonableness, in our setting the "relevant observables" are the complexes. The following equivalence equates two solutions if they possess the same complexes.

Definition 3.6 S *and* T *are* equivalent, *in notation* $\mathsf{S} \approx \mathsf{T}$, *if there exists a bijection f from nodes of* $\mathsf{graph}(\mathsf{S})$ *to nodes of* $\mathsf{graph}(\mathsf{T})$ *that preserves the species and such that $A[u](\sigma)$ and $B[v](\rho)$ are connected if and only if $f(A[u](\sigma))$ and $f(B[v](\rho))$ are connected.*

Notwithstanding the above revisions of Palamidessi's requirements, they turn out to be insufficient to exclude odd local-implementation protocols. In fact, our case is different than the one discussed in [45] where the dynamics of the calculi were fixed (those of pi calculus). In particular, the local-implementation protocol might completely redefine the dynamics of the encoded solution by tailoring the low-level reactions to the particular problem one wants to solve. For example, one might encode a κ-reaction by grabbing the reactants into one big molecule – that is, changing the degree of distribution – and then yielding the products – that is, re-establishing the degree of distribution. However, these encodings cannot be considered reasonable as much as maps that do not match Palamidessi's requirement of homomorphism.

Definition 3.7 *Let* $[\![\cdot]\!]$ *be an homomorphic encoding of (pre-)solutions and reactions in* κ *into (pre-)solutions and reactions in* nanoκ*. The encoding* $[\![\cdot]\!]$ *is* twinned *if, for every* $\mathsf{L} \to \mathsf{R}$

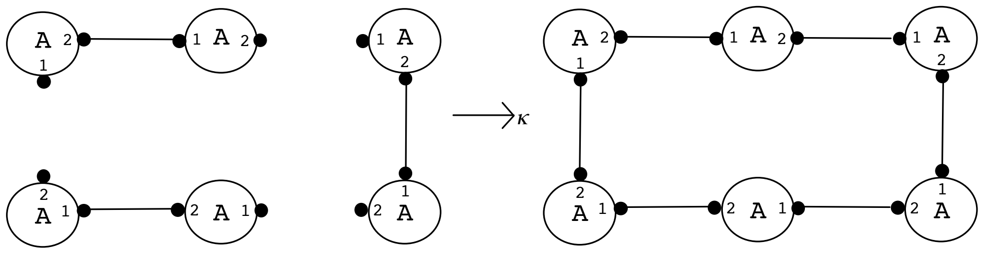
- *if it is a creation and* $[\![\mathsf{L} \to \mathsf{R}]\!]$ *contains a* nanoκ *destruction* $\mathsf{L}' \to \mathsf{R}'$ *then it also contains a* twin *creation* $\mathsf{R}' \to \mathsf{L}''$ *such that* L' *and* L'' *only differ for the values of fields;*
- *if it is a destruction and* $[\![\mathsf{L} \to \mathsf{R}]\!]$ *contains a* nanoκ *creation* $\mathsf{L}' \to \mathsf{R}'$ *then it also contains a* twin *destruction* $\mathsf{R}' \to \mathsf{L}''$ *such that* L' *and* L'' *only differ for the values of fields.*

Twinning guarantees that the local-implementation may undo some previous operation. The circularity may be avoided by yielding pre-solutions with different fields. Twinning also allows to localize the effects of protocols. Let us discuss the case of a κ creation $\mathsf{L} \to \mathsf{R}$. Then every destruction in $[\![\mathsf{L} \to \mathsf{R}]\!]$ has a corresponding creation restoring the bonds in the reactants. This means that if a destruction unbinds a molecule then the released molecule may be rebound, thus yielding either the previous complex with the same state – and the computation may diverge – or a previous complex with a different state. In this latter case, if the complex is too big with respect to those in R, then, there is a computation that either diverges or retains the complex (or a larger one). This because twinning will restore the complex if it is broken at some point. Therefore, in any case, the protocol is not semantically reasonable.

Definition 3.8 *Let A be a species with no field and two sites* 1 *and* 2*. An* homeotrimerization *is a κ-reaction:*

$$\begin{array}{l} A(1^x+2),A(1+2^x),A(1^y+2),A(1+2^y),A(1^z+2),A(1+2^z) \\ \longrightarrow\ A(1^x+2^u),A(1^v+2^x),A(1^y+2^w),A(1^u+2^y),A(1^z+2^v),A(1^w+2^z) \end{array}$$

that may be rendered as:



Theorem 3.1 *There exists no local-implementation protocol that is uniform, semantically reasonable and twinned for the homeotrimerization.*

Proof: Let S be a solution consisting of $2^m{*}3$ sticks $A(1^x{+}2),A(1{+}2^x)$ (we assume $m > 0$). This κ solution yields a stable solution T containing 2^m homeotrimeric complexes. By contradiction, let [![·]!] be a local-implementation protocol that satisfies the requirements of the theorem. We show that it is possible to construct an infinite derivation, thus yielding a contradiction because the S always terminates. We notice that the homeotrimerization is a creation; therefore in this case positive-direction set consists of creations and exchanges, while the negative-direction set consists of destructions and exchanges.

We analyze the protocol:

1. Initially every [![$A(\sigma)$]!] may arrange itself in order to participate to the homeotrimerization. Let us call *ready stick* such arranged sticks.

2. Then two ready sticks must be bound, thus making an homeodimeric complex.

Failure 1: Since the initial solution S consists of an even number of sticks, it is possible to obtain a solution with $2^{m-1} * 3$ homeodimeric complexes.

3. In order to avoid a deadlock, the protocol must admit bonds between two homeodimerics and then discharge one stick.

   Failure 2: The step 3 is not possible because the reaction discharging one stick is a destruction. By the twinning, the protocol must also admit a creation reconnecting the two sticks. Therefore, either one obtains the previous complex, thus yielding a divergent computation, or one obtain a complex with same bonds but different fields. So we are again in case 3. Eventually one gets either a solution with a complex of two homeodimerics and no destruction is possible, therefore it is not equivalent to [![T]!], or a divergent computation.

4. It remains the possibility for an homeodimeric to break the bond created in 2, thus releasing two sticks and breaking the symmetries (changing the fields). That is, while rolling back to 1, it is possible to mark the two sticks in "winner" and "loser", respectively. It is possible to obtain a solution where half sticks are marked as "winners" and half sticks are marked as "losers". Reactions of losers are frozen (otherwise no symmetry is broken). Then it is possible to build homeodimeric of winners and use losers to build homeotrimerics.

   Failure 3: It is possible to obtain a solution where a quarter of initial sticks is frozen (because they are losers). Therefore the protocol, in order to be semantically reasonable, must admit interactions between losers that yield homeodimerics. But this is not possible because they might be also performed before (losers do not know what happens in the context – the locality principle of the κ family), when homeodimerics of winners are built (and obtaining again a solution like 2). ∎

Our result has rather negative consequences. One for all is the impossibility of implementing a stochastic version of κ in nanoκ (or pi calculus) by preserving the distribution of rates (see [17]). This means that stochastic simulations must be done directly in κ [20].

Chapter 4

Leader election in the κ-family

In this chapter we investigate and compare the expressive power of various calculi within the κ family by studying the leader election problem in a symmetric network.

We work with κ calculus, $\mathtt{nano}\kappa$ calculus and a newly defined κ sub family called $\mathtt{ps}^{\mathrm{n}}$ within two different scenarios: a fully connected network and a ring network. The $\mathtt{ps}^{\mathrm{n}}$ has been defined mainly because we realized soon enough that both κ calculus and $\mathtt{nano}\kappa$ calculus are able to solve leader election in the two scenarios.
In order to point out which primitives were required by the two calculi to solve the problem we defined a very compact common sub-calculus, $\mathtt{ps}^2$, and then looked for the smallest extension of $\mathtt{ps}^2$ capable of solving leader election and still being a sub-calculus for κ calculus (respectively, $\mathtt{nano}\kappa$ calculus).
Our investigation led us to realize that the *synchronization degree*, i.e. the number of elements we allow to communicate simultaneously, has an important role in this context. Therefore an entire new family, called $\mathtt{ps}^{\mathrm{n}}$ and containing $\mathtt{ps}^2$, has been defined to study how different *synchronization degrees* affect expressiveness in the κ-family.

The results are presented bottom-up since we start from the $\mathtt{ps}^{\mathrm{n}}$ calculi and mainly focus on the synchronization degree issue, then consider various extensions and finally their relationship with the κ calculus and $\mathtt{nano}\kappa$ calculus calculi.

From a biological point of view our $\mathtt{ps}^{\mathrm{n}}$ calculi can be seen as good abstraction of polymeric structures – a grid of proteins of the same species with polymerizations that

rewrite connected proteins. On the other side the $\mathtt{ps}^{\mathrm{n}}$ calculi can be also seen as a graph rewriting framework with some restrictions on rules: this should not surprise as it is defined as a sub calculus of κ calculus which is essentially a particular restricted kind of graph rewriting. We demonstrate that it is not possible to elect a leader in polymers organized as a suitably large ring. This result entails that polymerizations rewriting at most n proteins are strictly less expressive than those rewriting $n + 1$ proteins. We also demonstrate that the leader election is solvable when polymerizations do connect proteins that are not directly connected or when they may flip connections. Finally we discuss how these results relate to the calculus of protein – the κ calculus –, of which our formalism happens to be a sub-calculus.
In the following we will mainly adopt a graph theoretical approach in the formalization and discussion of our study.

4.1 Introduction

A polymeric structure is typically a large molecule composed of many molecular units of the same type. The small molecular units interact and bind together so to form the larger polymer. Individually the molecular units are equivalent and cannot be distinguished therefore any macro behavior of the polymer must derive from the interaction of its composing units. We explore here what kind of interaction is needed for completely equivalent components to synchronize and reach a consensus in a symmetric scenario.

From a pure computer science point of view we compare different calculi suitable to describe polymeric structures and evaluate their synchronization capabilities.

When comparing the power of different calculi, their computational capability is only one, and the most obvious, property to evaluate. Another important aspect, when dealing with concurrent calculi for example, is indeed the synchronization capability of a calculus: for example whether independent processes in the calculus can reach an agreement or not.

We perform our evaluation by means of a typical problem testing the capability to reach a consensus: the leader election problem. The problem of leader election comes from the distributed systems field and was first posed by LeLann [35] who also gave the first solution. The problem tests the synchronization capability of a calculus and specifically it expresses the ability of a group of processes to reach an agreement without using a central coordinator.

More specifically the *symmetric* leader election problem requires that, starting from a configuration where each eligible component of the system is in the same state, a configuration is reached where exactly one component is in a special state *leader*, while all other components are in the state *lost*. The component in state *leader* at the end of the computation is called the *leader* and is said to be elected by the algorithm.

The main difficulty is to find an algorithm capable of breaking the initial symmetry of the system by reaching a configuration which is inherently asymmetric and where a leader can be always found.

This problem perfectly mimics the situation of a polymeric structure where a macro behavior has to be deterministically produced by means of interactions among completely equivalent subcomponents.

The rest of the chapter is organized as follows. In section 4.2 we introduce the *polymeric structures* calculus, its graphical notation and all the formal notions we will need. In section 4.3 the leader election problem along with the notions of symmetric network and electoral system is formulated within the setting of our calculus. Section 4.4 contains our main proofs, i.e. the non-existence of an electoral system for $\mathtt{ps}^{n}$ and the expressiveness hierarchy in the $\mathtt{ps}$ family by means of the separation between $\mathtt{ps}^{n}$ and $\mathtt{ps}^{n+1}$. In section 4.5 we discuss the existence of an electoral system for various cases: two calculi extending $\mathtt{ps}$ i.e $\mathtt{ps}_{c}$ and $\mathtt{ps}_{b}$.

4.2 Preliminaries

As already mentioned we favor the graph theoretical approach and therefore we present $\mathtt{ps}$ as a graph rewriting calculus. The biological interpretation is always at hand by remembering that each node represents a monomer, an edge represents a bond or interaction between two monomers and two or more monomers connected together represent a polymer structure.

We first introduce the syntax of the graph rewriting calculus we use, then discuss its graphical notation and finally give its semantics. We use a finite set of sites $\mathcal{S} = \{1, \cdots, l\}$, a finite set of fields $\mathcal{F} = \{1, \cdots, p\}$, a finite set of values $\mathcal{V}$ and a countable set of bonds $\mathcal{B}$. The full syntax is given in fig 4.2. The basic elements of this calculus are the *nodes*: each node has an *interface* and an *internal state* that correspond to maps $\mathcal{S} \mapsto \mathcal{B} \cup \{\varepsilon\}$ and $\mathcal{F} \mapsto \mathcal{V}$ respectively.

Sites are meant to be the interaction points for nodes and they may be either *bound* to other sites or *unbound*, i.e. not connected to other sites. The state of sites are defined by maps, called *interfaces* and ranged over by σ, ρ, $\cdots$. These interfaces are *partial functions* from $\{1, \cdots, l\}$ to the set of bonds plus a special empty value ε. A site i is bound with bond x in σ if $\sigma(i) = x$; it is unbound if $\sigma(i) = \varepsilon$.

For instance, if $|\mathcal{S} \models 3$, then $(2 \mapsto x, 3 \mapsto \varepsilon)$ is a possible interface for a node. In order to ease the reading, we write this map as $2^x + 3$ (the empty value ε is always omitted). This interface σ does not define the state of the site 1, which may be bound or not.

The internal state of a node depends on the values of its fields which are defined by maps as well, ranged over by u, v, $\cdots$. For instance, if $\mathcal{F} = 3$, then $[1 \mapsto 5, 2 \mapsto 0, 3 \mapsto 4]$ is a possible map. As before, we write this map as $1^5+2^0+3^4$. We will sometime use letters or short words and write map as $d^0 + locked^1 + count^4$ in order to ease the understanding of the fields' role. Since the set of values $\mathcal{V}$ is finite there are finitely many internal states.

A *pre-graph* is a sequence of nodes $[u](\sigma)$ where u and σ are not total functions: i.e. the information on the sites and fields status is not complete.

As expected we say two nodes in a graph G are *connected* if there is a path of bonds in between them. We define the *distance* between two connected nodes in a graph G as the number of bonds in the shortest path connecting the nodes.

Given a graph term G as specified above we also have a corresponding graphical notation, shown in Fig. 4.2, where all the information is retained.

We remark here that the syntactic representation is excessively intensional because, as shown in Fig. 4.2, syntactically different graph terms G and G' may give rise to the same graphical notation. More precisely each graphical notation corresponds to an equivalence

$a ::= [u](\sigma)$	(node)	$\lambda ::= \varepsilon \mid j \in \mathcal{B}$	(bond)
$G ::= \emptyset \mid a, G$	(graph term)	$u ::= \emptyset \mid z^{\gamma} + u$	(internal state)
$\sigma ::= \emptyset \mid i^{\lambda} + \sigma$	(interface)	$\gamma ::= j \in \mathcal{V}$	(value)
$i ::= j \in \mathcal{S}$	(site)	$z ::= j \in \mathcal{F}$	(field)

where u and σ are total in the node definition and with the operator $w + w'$ we assume the domains of maps w and w' are disjoint.

Figure 4.1: Syntax of $\mathtt{ps}^{\mathtt{n}}$ calculi

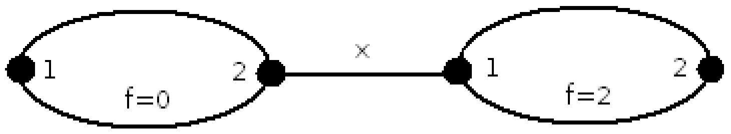


Figure 4.2: Graphical notation for both $G = [f^2](1^x + 2^\varepsilon), [f^0](1^\varepsilon + 2^x)$ and $G' = [f^0](1^\varepsilon + 2^x), [f^2](1^x + 2^\varepsilon)$

class of graph terms. This is due to the fact we are not using structural equivalence to keep the formalism as simple as possible.

Formally we can define exactly each graphical notation by means of the usual *nodes* and *edges* sets:

Definition 4.1 *Let $G = a_1, a_2, \cdots, a_l$ be a graph term and let v_i be the graphical notation for the single node a_i as specified in Fig. 4.2 with the exception that sites' status is removed. We define the G graphical notation, and we call it $\bar{G}$, by means of the couple of sets V and E where $V = \{v_i : i \in 1, 2, \cdots, l\}$ and*

$$E = \Big\{\Big\langle(v_i : s_1), (v_j : s_2)\Big\rangle : a_i \wedge a_j \text{ are bound in } G \text{ via sites } s_1, s_2\Big\}$$

A system in our framework is completely defined by a couple $< \mathcal{R}, G >$ where G is a graph with sites and internal values as defined in Fig. 4.2, while $\mathcal{R}$ is a set of rewriting rules of the form $L \to R$ where L and R are *pre-graphs*.

Semantics

The general schemata of a rewriting rule in our framework are the following

$$r_c : [u_1](\sigma_1), \cdots, [u_t](\sigma_t) \to [u'_1](\sigma'_1), \cdots, [u'_t](\sigma'_t), [v](\psi)$$

where u_i and σ_i are partial maps. In addition rules must respect the following conditions

1. the t nodes in the left hand side are all connected;
2. the creation of a fresh bond between nodes with distance greater than one is forbidden;
3. t is upper limited by a given integer n.

With this schemata it is clear that destructions of nodes are not permitted in our calculus. The second condition on rules concretely forbid the creation of bonds between nodes that are not directly connected, i.e. that do not already have a bond in between. For the given schemata this implies that the distance between two nodes may never be decreased as the effect of a rule. The last condition hides the fact that with the name `ps` we actually identify a family of rewriting calculi that differ on the number of nodes allowed to interact in the left hand side of rules: the specific calculus limited to synchronize at most n nodes will be referred to with the notation $\mathtt{ps}^{\mathrm{n}}$.

Given rule r as above the reduction relation over the graphs is the smallest relation satisfying the following rules:

$$Basic : \frac{}{[u_i + v](\sigma_i \circ \alpha_i + \rho) \xrightarrow{r,i,\alpha_i} [u'_i + v](\sigma'_i \circ \alpha_i + \rho)}$$

where α_i is an injective renaming on bond's names

$$Comp : \frac{a_1 \xrightarrow{r,1,\alpha_1} a'_1 \ \cdots \ a_t \xrightarrow{r,t,\alpha_t} a'_t}{G_0, a_{\pi(1)}, G_1, \cdots, G_{t-1}, a_{\pi(t)}, G_t \xrightarrow{r} G_0, a'_{\pi(1)}, G_1, \cdots, G_{t-1}, a'_{\pi(t)}, G_t}$$

where $\forall i, j\ \alpha_i(x) = \alpha_j(y)\ iff\ x = y$ and π is a permutation on $(1, \cdots, t)$.

Roughly speaking the *Basic* rule performs a matching between nodes of the graph and rules of the system. A transition $a \xrightarrow{r,i,\alpha_i} a'$ identifies the node a as a potential resource required by the left hand side of rule r. The basic transitions are actually used to build the *composite* ones which correspond to real transitions of the system. The *Comp* rule generates a transition labeled r whenever all the resources specified in the left hand side of the rule r are available in the graph, no matter how they are placed within the syntactic representation of the graph nor what the context is (i.e. the G_is). This is accomplished by looking if basic transitions, with compatible renamings α_i, are present for all the t nodes. For example, let $a = [1^3 + 2^1](1^z), b = [1^0](1^y)$, with a rule $\bar{r} = [2^1](1^x), [1^0](1^x) \rightarrow [2^0](1^x), [1^1](1^x)$ and a graph $G = G', a, b, G''$ the basic rule would produce transitions $a \xrightarrow{\bar{r},1,(z \mapsto x)} a'$ and $b \xrightarrow{\bar{r},1,(y \mapsto x)} b'$ but their renaming functions are not compatible therefore a composite transition for the rule $\bar{r}$ will not be produced: please note this is obviously correct since the rule requires the two nodes to be bound together while the nodes in G have different bond names (i.e. $z \neq y$).

Let's now keep a and $\bar{r}$ as above and consider $b = [1^0](1^z)$ and $G = G', b, G'', a$. Here we have that the two nodes are correctly bounded together but they are 'mixed' and 'misplaced' within G. Still, the *Comp* rule is able to correctly produce a transition $a, G'', b \xrightarrow{\bar{r}} a', G'', b'$ thanks to the permutation $\pi = (1 \mapsto 2, 2 \mapsto 1)$. A transition generated by rule *Comp* for rule r can be seen as a synchronization between the t nodes involved and leaves untouched any G_i filling the gaps between the t nodes in the syntactical term.

Automorphisms on graphs

We need to recall the concept of graph automorphism and properly adapt it to our graph framework before proceeding.

We will define the automorphism for the more intuitive graphical representation. Let $\bar{G} = (V, E)$, where V denotes the set of nodes and E the set of edges between nodes, be the graphical notation for some graph term G. We define a *type function* t_E on edges as follows:

$$t_E(x) = \left\langle (v_i; s_1), (v_j; s_2) \right\rangle$$

where $v_i, v_j \in V$ are the nodes connected by x, and $s_1, s_2 \in N$ the specific sites used by the edge $x \in E$. The function t_E therefore assigns to each edge $e \in E$ the couple of nodes, along with the sites involved, connected by e itself. We also need a type function t_V on nodes

$$t_V(z) = [\gamma_1, \cdots, \gamma_p] \quad \gamma_i \in \mathcal{V}$$

returning a tuple of values that represents the node's complete internal state.

Definition 4.2 (Graph automorphism) *An automorphism on $\bar{G}$ is a pair $\delta = \langle \delta_V, \delta_E \rangle$ such that $\delta_V : V \longmapsto V$ and $\delta_E : E \longmapsto E$ are permutations that preserve the type of edges and the type of nodes, i.e. for any $e \in E$ if $t(e) = \left\langle v_i; s_1, v_j; s_2 \right\rangle$ then $t(\delta_E(e)) = \left\langle \delta_V(v_i); s_1, \delta_V(v_j); s_2 \right\rangle$ and for any $v \in V$ $t_V(v) = t_V(\delta(v))$.*

It is not difficult to show that the composition of automorphisms is still an automorphism. Given automorphisms δ and δ' the composition is defined component-wise $\delta \circ \delta' = \left\langle \delta_V \circ \delta'_V, \delta_E \circ \delta'_E \right\rangle$. The identity automorphism is defined as the couple of identity functions on V and E, i.e. $id = \langle id_V, id_E \rangle$. The set of automorphisms on $\bar{G}$ with the composition operation forms a group.

The *orbit* of $v \in V$ generated by the automorphism δ is defined as the set of nodes in which the various iterations of δ map v:

$$O_\delta(v) = \{v, \delta_V(v), \delta_V^2(v), \cdots, \delta_V^{h-1}(v)\}$$

where δ_V^i is the composition of δ_V with itself i times, and h is the least number of iterations such that $\delta_V^h = id_V$. The set of all the orbits generated by δ forms a partition of V: we call it *orbit of* δ and we write $O(\delta)$.

An automorphism δ is said *well-balanced* if all the sets in $O(\delta)$ (i.e. all its orbits) have the same cardinality. We call such cardinality *degree* of δ.

4.3 Leader election, electoral system: formal notions

In this section we formalise the leader election problem in the setting of our `ps` calculus. We point out here that the constraint requiring the left hand side of rules to be connected perfectly simulates the communication issues between elements in a distributed system: bonds represent communication channels and a synchronization between nodes may not take place if they cannot communicate. In other words we will use the connectedness as an abstraction for the existence of a communication path between components.

Leader Election in Distributed Systems

The leader election problem comes from the distributed systems field and the generic problem consists in ‘electing a leader’ among several processing independent units. By the term ‘distributed system’ it is usually meant an *interconnected* collection of *autonomous* machines. The machines are referred to as the *nodes* of the distributed system and in order to be qualified as autonomous each node should have its own private control. To be qualified as interconnected, the nodes must be able to exchange information. A computation is performed by the distributed system by means of the interactions between its nodes. The computation of a distributed system should be obviously evaluated at the system level, yet every single node of the system is fully qualified as an independent computing device.

The leader election problem has been successfully represented in process algebra. The first formalisation is due to Bougé [7] who formalised the problem in symmetric networks for CSP. The notion of leader election was later formalised in a similar way by Palamidessi [45] for the $\pi-$calculus. In contrast with process algebra where the behaviours are mostly encoded in the processes, the dynamics and behaviours of polymers in our `ps` calculus are expressed in the form of rewriting rules. Therefore we can’t straightforwardly reuse the existing formalisation and need to provide a new one that suits the graph-rewriting nature of `ps`.

The leader election problem requires that, starting from a configuration where each eligible component of the system is in the same state, a configuration is reached where exactly one component is in a special state *leader*, while all other components are in the state *lost* and can no longer become leader. The component in state *leader* at the end of the computation is called the *leader* and is said to be elected by the algorithm.

It is not required that the so called *eligible component* be a basic element (i.e. a single node of the graph) of the system but it could as well be defined as a more complex structure with some specific property that let it regard as an *autonomous unit* of the system. Yet, and without loss of generality we will map eligible components into single nodes in order to ease the reading and the formalisation of few definitions. A *network* is informally a composition of such eligible components or separate units of the system.

Observation relation

There are several ways to define an observational predicate given a system $< G, \mathcal{R} >$, we could for example be interested in monitoring the development of a special structure or we may want to know if a special internal state has been reached by some node or finally we could monitor the firing of some special rule during the system computation. We opt for the latter as it perfectly suits our problem.

Given a system with rules defined by the set $\mathcal{R}$ the set of observable reactions Obs is defined as a subset of $\mathcal{R}$.

Definition 4.3 (Computation) *Given a system $S = < \mathcal{R}, G >$ a computation C of S is a (finite or infinite) sequence of transitions $G = G_0 \xrightarrow{\omega_1} G_1 \xrightarrow{\omega_2} \cdots$, with $\omega_i \in \mathcal{R}$. A computation is called* maximal *if it cannot be extended: either C is infinite or it is of the form $G_0 \xrightarrow{\omega_1} G_1 \xrightarrow{\omega_2} \cdots \xrightarrow{\omega_h} G_h$ where $G_h \not\rightarrow$.*

Definition 4.4 *Given a computation $C = G_0 \overset{\omega_0}{\rightarrow} \cdots G_l \ {}^{\omega_l} \rightarrow \cdots$ we say $\omega_i \in C$ if there exists $G_i \overset{\omega_i}{\rightarrow} G_j$ in C. We define the* observables *of C to be the multiset $Obs(C) = \biguplus_i \{\omega_i : \omega_i \in C$ and $\omega_i \in Obs\}$ where the operator $\uplus$ stands for the* multiset union.

Definition 4.5 (Network) *A network $\mathbf{N}$ of size k is a couple $(R, \langle a_1, \ldots, a_k \rangle)$, where R is a set of rules and $\langle a_1, \ldots, a_k \rangle$ a sequence of nodes. Each a_i is said* component *of the network $\mathbf{N}$.*

Informally with the network concept we intend to explicitly represent the distribution in the system of the various components forming the system itself: each of them is given an index that could be seen as a positional information in the network topology. The only difference between a network $\mathbf{N} = (R, \langle a_1, \ldots, a_k \rangle)$ and a system $S = < R, a_1, \ldots, a_k >$ is indeed just the added index information for each node.

In our ps calculus we do not admit structural congruence on terms and therefore graph terms composed by the very same nodes but with different orders are considered distinct. We exploit this fact and consider a system S as above as a network where each node is naturally assigned its term position number as the index of the network: e.g. in graph a, b, c node a has position 1 while nodes b and c have position 2 and 3 respectively. That means that for all practical purposes a network is a system in which we do care about the nodes position in the graph term G. Obviously networks inherit all the notions defined for the systems, e.g. computation and observables. We say δ is an automorphism on the network $\mathbf{N} = (\mathcal{R}, G)$ if it is an automorphism on $\bar{G}$. Let $\Phi_{\mathbf{N}} = \{id, \delta_1, \cdots, \delta_k\}$ be the set of all the automorphisms on the network $\mathbf{N}$.

Definition 4.6 (Symmetric network) *A network* $\mathbf{N}$ *of size* m *is symmetric if it admits a well-balanced automorphism* $\delta \neq id$*. It is* fully symmetric *if* δ *has degree equal to* m*, i.e. if all the nodes are identical but for the bond names.*

It could be shown that any well-balanced automorphism corresponds to a graphical rotational symmetry on the graph itself[1]. Intuitively a network is symmetric if it is possible to spatially rearrange the nodes, without altering the edges, of its graphical notation so to see a rotational symmetry. A well-balanced automorphism with orbits of size k corresponds to a so called *k-fold* rotational symmetry.

In order to define an electoral system we need to specify how the system communicates to the external world that the election of a leader has been completed. We do that by means of a special reaction ω_{out}, with $\omega_{out} \in Obs$. We actually assume Obs to contain exactly ω_{out} as the only observable of the system and, without loss of generality let ω_{out} be triggered by a special internal state of a single component of the network which becomes

[1] Actually on its graphical notation.

the *leader* as the effect of the reaction itself. Finally lets assume ω_{out} can be fired only once for each component and never for a component which has been marked as *defeated* by means of its internal state.

Definition 4.7 (Electoral system) *A network* $\mathbf{N}$ *is an electoral system if any maximal computation* C *of* $\mathbf{N}$ *is finite and contains only one occurrence of transition* ω_{out}.

A network is therefore an electoral system if it never diverges and eventually reports a unique leader among its components.

4.3.1 Scenarios

We will consider the leader election problem in two different scenarios corresponding to two different topologies. While the sites and fields of the nodes will be properly defined case by case, each node will have at least a field d that is set to 1 if the node gets defeated during the selection of the leader. Initially all the d fields are set to 0.

Fully connected network

This is the standard scenario for leader election as studied in [45] and we summarize it with the following parameters:

- no restriction on the topology of the network: we will assume fully connected networks;
- the size of the network is supposed to be known: that means the number of the nodes is fixed and known;
- the declaration of the leader will be announced by the leader itself for simplicity.

A system $< \mathcal{R}, a_1, \cdots, a_m >$ is a fully connected network if for each $i, j \in 1, \cdots, m$ there exist s_1, s_2 and x such that $a_i = (u_i)[s_1^x + \sigma_i]$ and $a_j = (u_j)[s_2^x + \sigma_j]$.

Ring network

The second scenario is more restrictive and thus more interesting as it requires the network to be organized in a ring. A component of the ring is supposed to interact only with its left-hand and right-hand neighbors. Typically the algorithms which assume that all the components of the network can interact directly will no longer work. A system $< \mathcal{R}, a_1, \cdots, a_m >$ is a ring connected network if for each node a of the system there exist exactly two distinct indexes $i, j \in 1, \cdots, m$ such that a is directly connected to a_i and a_j.

4.4 Non existence of a symmetric electoral systems in $\mathtt{ps}^{\mathrm{n}}$ for ring networks

We show here that, for polymers organized as symmetric ring networks, the leader election problem is not solvable in $\mathtt{ps}^{\mathrm{n}}$ when the network size is big enough with respect to n.

In this section we favor the graph theory point of view for it provides a more intuitive approach to address the symmetry related issues of the problem.

We already said that an automorphism δ gives rise to a partition on the nodes of a network by means of its orbits. If δ is also well-balanced with degree d than the partition is composed by exactly m/d orbits. Intuitively this means it is possible to see the network as composed by d completely symmetric parts, and they also form a partition on the nodes.

Each symmetry of a network is represented by a well-balanced automorphism and when the network is fully symmetric there is exactly one automorphism δ_i of degree d_i for each d_i positive divisor of the network size.

We remind n corresponds to the maximum number of nodes a rule can specify in its left hand side. Moreover, in $\mathtt{ps}^{\mathrm{n}}$ we may synchronize only connected components. In a ring network this specifically means it is possible to synchronize only chains of at most n nodes.

Fact 1 *In $\mathtt{ps}^{\mathrm{n}}$ a rule may synchronize nodes whose distance is at most $n - 1$.*

Fact 2 *Let δ be a well-balanced automorphism with degree d on a ring network of size m. Nodes of the same orbit are equidistant in the network: the distance between a node b and $\delta(b)$ is exactly m/d.*

The idea behind the proof is that, under certain conditions, whenever a rule takes place in a symmetric ring network, it can be applied again on each of the $d-1$ symmetric portions of the network for some automorphism δ with degree d thus restoring the symmetry induced by δ itself after $d-1$ steps. The process can be repeated for any rule and a maximal computation which preserves the δ symmetry is produced. In such maximal computation the election of a leader never succeeds either because no node is declared the winner or, if a node becomes the leader by means of transition ω_{out} then any node in the same orbit wrt δ may do the same.

Lemma 4.1 *Let* $\mathbf{N} =< \mathcal{R}, G >$ *be a symmetric ring network of size m and δ one of its automorphisms with degree d such that $n \leq m/d$, then for any transition r on G there exists a sequence of $d-1$ transitions r that reestablish the symmetry induced by δ on the network.* **Proof:**

Since we assumed $n \leq m/d$ it immediately follows from Facts 1 and 2 that any rule in $\mathcal{R}$ may not synchronize two nodes symmetric wrt δ, i.e. two nodes in the same orbit of δ. This guarantees that any rule modifies at most one node for each orbit of δ and leaves untouched the others.

Let's now consider a rule r with arity $n' \leq n$. If a transition of type $\xrightarrow{r}$ is executed than it involves exactly n' nodes of the network and they are all connected in a chain. The effect of a single transition on the the network is, in general, that symmetry is broken: in the worst case the internal state of each node involved is modified so to obtain a distinct internal state on a network level. So far δ no longer represents a symmetry on the network.

Let $a_i, \cdots, a_j$ be the chain of nodes involved by the first transition. Consider node a_i, it has been modified according to transition r and therefore it is no longer equivalent with the nodes of its orbit. If we could modify the other $d-1$ nodes in the very same way we would reestablish the equivalence for that orbit.

This is indeed possible by applying $d-1$ times the rule r on the network: for symmetry of δ we have that another transition of type $\xrightarrow{r}$ can be applied on the symmetric chain of nodes $\delta_V(a_i), \cdots, \delta_V(a_j)$ and in general on the chain $\delta_V^h(a_i), \cdots, \delta_V^h(a_j)$ with h ranging over $1, \cdots, d-1$. After those $d-1$ transitions, δ regain the status of a well-balanced automorphism and its symmetry on the network is preserved.

This is no longer true if $n > m/d$: in this case a rule could have arity $n' > m/d$ and any application of such a rule would affect (at least) two nodes of a same orbit. It should be clear that in general the two nodes will not be symmetric after the rule application and we cannot reapply the rule $d-1$ times simply because there are no d symmetric chains of size n' in the network: indeed $n' \cdot d$ is greater than the network size! □

Theorem 4.1 *Let* $\mathbf{N} = (\mathcal{R}, < a_1, \cdots, a_m >)$ *be a symmetric ring network, if* $\mathbf{N}$ *admits a well-balanced automorphism* $\delta \neq id$ *with degree* d *such that* $n \leq m/d$*, then* $\mathbf{N}$ *is not an electoral system for* $\mathtt{ps}^{\mathrm{n}}$. **Proof:** We have a network $\mathbf{N}$ symmetric wrt some automorphisms δ, we may always produce a maximal computation on $\mathbf{N}$ that does not resolve leader election:

1: $\mathbf{N} \nrightarrow$ if no transition is possible and a leader is missing: $\Rightarrow$ failure!

2: $\mathbf{N} \xrightarrow{r} \mathbf{N'}$ a transition is possible

There are two sub cases:

a: $r = \omega_{out}$ – A node a_i has been elected as leader

For symmetry of δ we may perform a transition r also on the symmetric node $\delta(a_i)$ producing another leader $\Rightarrow$ failure!

b: $r \neq \omega_{out}$ – We have no leader and the network symmetry is broken

For Lemma 4.1 we may undertake a computation $\mathbf{N'} \xrightarrow{r} \mathbf{N'_1} \xrightarrow{r} \cdots \xrightarrow{r} \mathbf{N'_{d-1}}$ such that $\mathbf{N'_{d-1}}$ is symmetric wrt the automorphism δ

$\Rightarrow$ we have restored the initial conditions: failure or infinite computation preserving δ and no leader!

□

Please note that if m is prime then the theorem holds only for $n = 1$ no matter how big m is. In fact it can be shown that for any ring network of prime size $\mathtt{ps}^2$ can resolve leader election.

Since conditions of Theorem 4.1 depend on the network size m it is clear that by choosing a suitably large network we can always fulfill the theorem's requirements. That is, for any $\mathtt{ps}^{\mathrm{n}}$ there exist a polymer organized as a ring with size big enough so that leader election is not solvable with $\mathtt{ps}^{\mathrm{n}}$.

Hierarchy on the $\mathtt{ps}^{\mathrm{n}}$ family

We exploit here our main result to show that $\mathtt{ps}^{\mathrm{n}}$ is strictly less expressive than $\mathtt{ps}^{\mathrm{n+1}}$. The idea is to take a ring network of a specific size and show it is an electoral system only for one of the calculi.

Corollary 4.1 *Let* $\mathbf{N} =< \mathcal{R}, G >$ *be a symmetric ring network of size* $m = 2n$. *Such a network is not an electoral system for* $\mathtt{ps}^{\mathrm{n}}$. **Proof:** Clearly 2 is a divisor of m and therefore there exists an automorphism δ of degree $d = 2$ on the network $\mathbf{N}$. Since the conditions of theorem 4.1 are satisfied

$$n \leq m/d \ = \ n \leq 2n/2 \ = \ n \leq n$$

we have that for such a ring network leader election is not solvable with $\mathtt{ps}^{\mathrm{n}}$. ▮ □

To have the separation we have to demonstrate that such a network is an electoral system for $\mathtt{ps}^{\mathrm{n+1}}$.

Theorem 4.2 *Let* $n \in \mathbb{N}$ *and* $\mathbf{N} =< \mathcal{R}, G >$ *be a symmetric ring network of size* $m = 2n$. *Such a network is an electoral system for* $\mathtt{ps}^{\mathrm{n+1}}$. **Proof:** The proof consists in the description of an easy protocol solving this specific scenario. The only remark we make is that the solution requires the network size to be known.

We have a ring network of size $m = 2n$ and we may use rules with at most $n + 1$ connected nodes in their left hand side. Initially all the nodes are eligible and the very first rule takes a chain of $n + 1$ eligible nodes and marks n of them as losers. We opt to leave the eligible one at the clockwise end of the chain. Obviously such a rule may be performed only once since only n eligible nodes are left. A second rule is triggered by a chain of $n + 1$ nodes where only one, at an end of the chain, is marked as loser and marks all the eligible nodes but one as losers. The node not marked as loser is instead marked as the leader.

The algorithm is quite easy since it is composed by only two rules and therefore it is immediate to see it always produce a single leader in such a symmetric network. ▌ □

Corollary 4.2 *Let $n \in \mathbb{N}$,* $\mathtt{ps}^{\mathrm{n}}$ *is strictly less expressive than* $\mathtt{ps}^{\mathrm{n+1}}$. **Proof:** From theorems 4.2 and 4.1 we have that for each $n \in \mathbb{N}$ there exists a symmetric ring network of size $m = 2n$ which is an electoral system for $\mathtt{ps}^{\mathrm{n+1}}$ but not for $\mathtt{ps}^{\mathrm{n}}$. ▌ □

The separation has been done with a generic n therefore it leads to an expressiveness hierarchy on the whole ps family.

4.5 Existence of a symmetric electoral system in $\mathtt{ps}^2$ for fully connected networks

Here we show an algorithm that, assuming known the network size, solves leader election problem in fully connected polymers using only binary polymerizations.

The algorithm idea is fully explained but we leave out few formal details.

Let m be the network size, the generic protein in its initial configuration is described by

$$[d^0 + l^0](1^{x_1} + 2^{x_2} + \cdots + m - 1^{x_{m-1}}) \tag{4.1}$$

where bonds x_i connect the protein with all the others $m-1$ composing the structure, field l marks the protein as leader when its value is equal to 1 and field d marks the protein as defeated if its value is set to 1.

In this scenario we just need a very simple (schema of) polymerization in order to elect a leader. The algorithm idea, not new at all, is to let eligible components of the network *fight* each other two at a time until a single one remains and is therefore declared the leader. A ‘fight’ consists in a synchronization between two proteins with their fields d set to 0 where one protein becomes defeated by setting its d field to 1.

This simple idea is realized by the following schema that generates all the ‘fight’ polymerizations we need

$$fight_{i,j} : [d^0](i^x), [d^0](j^x) \rightarrow [d^1](i^x), [d^0](j^x) \qquad i, j \in 1, \cdots, m-1 \tag{4.2}$$

We leave out the correctness proof for it is not difficult to see that in a system composed by m proteins, defined as specified in (4.1), together with the set of $fight_{i,j}$ polymerizations after any maximal computation we are left with a structure where a single protein has internal state $d^0 + l^0$: clearly no polymerization from the fight schema can happen anymore and a leader has been found.

There are several ways to proceed once a leader is found. Probably not the most efficient we illustrate a simple one. We have to make sure the leader is aware of being the leader and that all the defeated proteins are told who the leader is as well.

We may, for example, add a *counter* field c to each protein telling how many proteins were directly or indirectly defeated by the protein itself. The rough idea is that every time a protein win a *fight* synchronization we add the c value of the defeated protein plus one to the c value of the winner. To integrate this idea the *fight* schema must be extended to the following one:

$$[d^0 + c^l](i^x), [d^0 + c^{l'}](j^x) \rightarrow [d^1](i^x), [d^0 + c^{l+l'+1}] \text{ with } l + l' \leq m-2 \wedge i, j \in 1, \cdots, m-1$$

The leader is then recognized by looking the internal state of proteins with the following:

$$\omega_{out} : [d^0, l^0, c^{m-1}] \rightarrow [d^0, l^1, c^{m-1}] \quad \text{leader found!}$$

A protein becomes the leader only if it has defeated all the others $m - 1$.

Once a leader is recognized, it can contact all the defeated proteins letting them know about its identity thus completing the algorithm. We omit the reaction schema for the leader notification protocol.

4.6 Existence of a symmetric electoral system in $\mathtt{ps}_b^2$ and $\mathtt{ps}_c^3$ in ring networks

We introduce now two variants of $\mathtt{ps}^{\mathrm{n}}$ and show how to solve leader election with such calculi. The first variant is obtained by adding a new rule schema that let an existing bond to be flipped from one node to another, while the second one is obtained by relaxing the condition on bonds creation between non adjacent nodes.

Definition 4.8 *The* $\mathtt{ps}_b^{\mathrm{n}}$ *calculus is defined as* $\mathtt{ps}^{\mathrm{n}}$ *with the addition of a new kind of polymerization, called* bond flip*, defined by the following schema*

$$bf : [u](a^x + c^y + \rho'), [v](b + d^y + \psi') \longrightarrow [u'](a + c^y + \rho'), [v'](b^x + d^y + \psi')$$

In $\mathtt{ps}_b^{\mathrm{n}}$ we may write $[1^0](1^x + 2^y), [1^0](1^y + 2) \longrightarrow [1^1](1 + 2^y), [1^1](1^y + 2^x)$ meaning that the bond x *migrates* from the first protein to the second: the other end of the bond remains untouched. We notice that a bond flip rule never modify the connectedness of a structure and it does not create new bonds.

Definition 4.9 *The* $\mathtt{ps}_c^{\mathrm{n}}$ *calculus is defined as* $\mathtt{ps}^{\mathrm{n}}$ *but polymerizations are now allowed to create bonds between proteins with distance greater than one (i.e. not directly connected).*

Solving Leader Election in $\mathtt{ps}_b^2$

As in the fully connected scenario we still use a *fight* schema to determine the loser between directly connected candidates but we need a protocol to determine the loser between non adjacent ones (not directly connected).

Informally we have a two phase algorithm. The first phase consists in fights between adjacent proteins as seen above. When the first phase is over we are generally left with a ring where candidates are not adjacent[2]. The second phase will try to connect any isolated candidate to the first candidate encountered following the ring in a counter clockwise way in order to let them fight until only one protein is left as a candidate.

[2]With the only exception of the rare case the first phase lead directly to a single candidate.

The generic protein for this scenario in its starting configuration is defined as

$$[d^0 + l^0](1^x + 2^y + 3^\varepsilon + 4^\varepsilon)$$

Fields d and l have the same meaning as seen before. Sites 1 and 2 are meant to connect the protein to its clockwise and counter clockwise neighbors respectively, while site 3 and 4 will be used to connect isolated candidates using bond flipping. We precise now that a bond connected to a site 3 will always have a site 4 on the other end, moreover the bond's end on site 3 will be fixed while we will be flipping the bond's end on site 4.

The following two rules are responsible for the first phase of the algorithm, i.e. the fights between adjacent candidates

$$\textit{Adjacent fight}: \quad \begin{cases} [d^0](1^x), [d^0](2^x) \longrightarrow [d^1](1^x), [d^0](2^x) \\ [d^0](1^x), [d^0](2^x) \longrightarrow [d^0](1^x), [d^1](2^x) \end{cases}$$

Second phase: bond flipping and non local fights

As already said, after the first phase we are generally left with a ring network where two or more candidates are not adjacent. In order to let these isolated candidates interact we need to create a bond between them[3], but since we are forbidden to directly create bonds between non adjacent candidates we first create a special bond between an isolated candidate and its counter clockwise (defeated) neighbor and then we use the *bond flip* rule to *flip* the neighbor's end of the special bond to its counter clockwise neighbor. After the first flip the isolated candidate is therefore connected to a third node of the network and they can interact. The idea is simply that if the third node is also a candidate a fight interaction marks one as the loser otherwise the special bond's end on the third node is flipped again to the next node in counter clockwise order and so on until another isolated candidate is found.

The special bond creation and flipping is realized with the following rules

$$\begin{aligned} cre : [d^0 + l^0](2^x + 3), [d^1](1^x + 4) &\longrightarrow [d^0 + l^0](2^x + 3^y), [d^1](1^x + 4^y) \\ flipping : [d^1](2^x + 4^y), (1^x + 4) &\longrightarrow [d^1](2^x + 4), (1^x + 4^y) \end{aligned}$$

[3]We are working with ps_b^2 so only two proteins at a time may synchronize and they must be connected.

We may safely assume that the creation of the special bond occurs only when both the neighbors are marked as defeated even if the rule *cre* above checks only one. This can be easily accomplished by means of two rules checking the neighbors status and two fields storing this information.

The non local fight between ‘isolated’ candidates connected by means of a special bond is straightforward

$$\textit{Non local fight:} \qquad \begin{cases} [d^0](3^y), [d^0](4^y) \longrightarrow [d^0](3), [d^1](4) \\ [d^0](3^y), [d^0](4^y) \longrightarrow [d^1](3), [d^0](4) \end{cases}$$

To keep the protocol easy to understand, we are not interested in efficiency right now, we destroy the flipped bond after every fight between non adjacent candidates. It should be clear that an isolated candidate will always start the bond flipping protocol by creating the special bond with its neighbor unless the candidate itself has been marked as the leader.

Leader recognition

A nice property of the algorithm described here is that it does not need to know the network size, i.e. it is size independent. This property comes from the ring structure of the network and the fact that a node may interact with all the others following the ring order by flipping a single bond.

In order to recognize a node as the leader we must check it is the only one still eligible in the network, i.e. that all the others have been marked as *defeated*. Let’s consider then a network configuration where a single candidate is still eligible: we have that the second phase protocol would create a bond between the candidate and its neighbor in the counter clockwise direction, the bond would be *flipped*, in the same counter clockwise direction, until another eligible candidate is found. Since there is only a single eligible candidate the bond would be flipped through the whole ring until it reaches the initial node: actually if self connections are allowed in our calculus we would end up having a loop bond on the single candidate otherwise the bond would connect the candidate with its neighbor in clockwise direction. In both cases the flipping protocol would stop. Let’s assume we do not admit self connections, the following rule recognizes the leader:

$$\omega_{out} : [d^1](2^x + 4^y), [d^0 + l^0](1^x + 3^y) \xrightarrow{\omega_{out}} [d^1](2^x + 4^y), [d^0 + l^1](1^x + 3^y)$$

Once a leader has been elected, it can notify the others about its identity using again a simple protocol that flips a bond through the network.

Since we never mentioned the network size in the algorithm description it should be clear it is size independent.

Solving Leader Election in $\mathtt{ps}_c^3$

As in the solution in $\mathtt{ps}_b^2$ we have a two phase algorithm. The main difference is we no longer have bond flipping but we are allowed to create new bonds between nodes with distance greater than one. Concretely this allows to decrease the distance between nodes in the ring in a similar way we did before using bond flipping.

The first phase of the algorithm is identical to the one for $\mathtt{ps}_b^2$ while the second phase is similar but we use a ternary rule and two special bonds. An isolated candidate is first connected to its neighbor with distance equal to two, then a fight takes place if the neighbor is a candidate as well otherwise a second bond is created connecting the isolated candidate and the neighbor that before the first bond creation had distance three with the isolated candidate. This is possible only because the first bond creation decreased by one the distance between the isolated candidate and any neighbor with distance greater than one. Again, after this second special bond is created, the first bond is deleted and a fight takes place if the newly connected neighbor is an eligible candidate otherwise the process iterates until another isolated candidate is found. It should be easy to see this is very similar to the previous solution. For everything else the algorithm is identical to the one for $\mathtt{ps}_b^2$.

Clearly the creation rule cannot perfectly simulate the bond flipping, and this is the reason we are using $\mathtt{ps}_c^3$ and not $\mathtt{ps}_c^2$.

4.7 Conclusion

We have considered a new family calculi suitable to describe polymeric structures and analyzed its synchronization capabilities. This family, called $\mathtt{ps}^{\mathtt{n}}$ has been defined ad hoc and its *smallest* (less expressive) calculus $\mathtt{ps}^2$ is a subset for both the κ calculus and nanoκ calculus.

The various calculi have been compared by assessing their capability in solving the leader election problem and we considered two different scenarios: a fully connected network and a ring network. The first scenario was rather unhelpful for any comparison since we prove that the leader election is solvable in all the calculi considered by providing a solving algorithm for the $\mathtt{ps}^2$ calculus: this holds because $\mathtt{ps}^2$ is a subset for both the κ calculus and nanoκ calculus, and is also the less powerful representative of the $\mathtt{ps}^{\mathtt{n}}$ family.

The ring network scenario is more interesting. We have that the $\mathtt{ps}^{\mathtt{n}}$ family is unable to solve the leader election in general but interestingly any given calculus $\mathtt{ps}^{\mathtt{m}} \in \mathtt{ps}^{\mathtt{n}}$ can solve the leader election when the ring network size is less than $2m - 1$. This is due to the fact that in $\mathtt{ps}^{\mathtt{m}}$ we can synchronize m elements at a time. A corollary of these result is that the $\mathtt{ps}^{\mathtt{n}}$ retains a totally ordered hierarchy wrt the expressive power: $\mathtt{ps}^{\mathtt{m}+1}$ is strictly more powerful than $\mathtt{ps}^{\mathtt{m}}$.

From a biological point of view our main proof in theorem 4.1 entails that the $\mathtt{ps}^{\mathtt{n}}$ calculus is not sufficient to describe a polymeric structure of any size whose components must interact to deterministically produce a macro behavior. This is the case indeed only for polymers whose size is bounded with respect to n, where we remind that n is the maximum number of monomers that may interact simultaneously.

We have then showed that $\mathtt{ps}^2$ extended with the bond-flipping primitive, i.e. $\mathtt{ps}^2_b$, and $\mathtt{ps}^3$ with no constraint on bonds creation, i.e. $\mathtt{ps}^3_c$, are enough to let monomers reach a consensus. These results state the expressive power of the bond-flipping operation and of bond creations that disregard any constraint. It is worth to notice that the algorithm solving the problem for $\mathtt{ps}^2_b$ does not depend on the network size while all the other considered in this chapter do.

The formalization we gave of the leader election problem is similar to the one given by Palamidessi in [45] and our non-existence result has a similar approach. The main difference is that we had to deal with a much finer grained symmetry with respect to the full symmetry used by Palamidessi. Our `ps` calculus is indeed able to break the initial symmetry of the system but only to a certain degree. That is, full symmetry is easy to break with `ps` but there is typically a symmetry induced by a well-balanced automorphism which is unbreakable. This result depends on the degree of the automorphism and the number of nodes allowed in the left hand side of rule.

Chapter 5

The Expressive Power of Synchronizations

In this chapter we leave completely our bio inspired world and present some results into Theoretical Computer Science. This study has been actually inspired by our previous work in chapters 3 and 4 and precisely by the insights on synchronizations among more than two processes but the connection with bio inspired calculi goes no further. We are interested in studying what we could call *multi synchronization mechanisms* within a well established framework.

By the term *synchronization* we refer to a mechanism allowing two or more processes to perform actions at the same time. In this chapter we study the expressive power of synchronizations gathering more and more processes simultaneously. We demonstrate the non-existence of a uniform, fully distributed translation of Milner's CCS with synchronizations of $n + 1$ processes into CCS with synchronizations of n processes that retains a "reasonable" semantics. We then extend our study to CCS with more liberal synchronizations allowing a process to perform both inputs and outputs at the same time. We demonstrate that synchronizations containing more than three input/output items are encodable in those with three items, while there is an expressiveness gap between three and two.

Finally we compare the introduced synchronization mechanisms with various choice operator.

5.1 Introduction

Process calculi propose several different synchronization mechanisms. In CCS and pi calculus, the synchronization is between two processes, one sending a message and the other receiving it [38, 42]. In CSP, the synchronization is among all the processes that share a common channel name [8]. Join calculus has a programmable synchronization mechanism – the *joint inputs* –, which allows one to define the channels whose messages must be handled simultaneously [26]. Other calculi use joint input mechanisms, such as smooth orchestrators [32] – an extension of asynchronous pi calculus to program web services orchestrations – and strand algebras – a recent formalism proposed for DNA computing [14].

Already in the first contribution about the join calculus, Fournet and Gonthier presented an encoding of the generic synchronizations to basic ones consisting of binary synchronizations [26]. There are two problems with Fournet and Gonthier's encoding. First, their encoding introduced divergence and, in facts, it was proved correct with respect to coupled simulation, which is unsensible to divergent computations. For this reason, the encoding is not "reasonable" in the sense of Palamidessi [45]. Second, the encoding cannot be considered "truly distributed" because it relies on the locality principle that constrains co-defined channels to be co-located.

It is folklore in the Concurrency Theory community that some expressiveness gap between the different forms of synchronizations must exists. For example, in a calculus *à la* CCS without mixed choice it is not possible to elect one leader in a fully connected network of (symmetric) processes without introducing divergence [45]. On the contrary, if one extends CCS with a prefix $[a_1, a_2].P$, which enables P if there are two outputs on a_1 and a_2, then leader election has the following simple solution. Let m be the processes that wants to elect a leader, and let $h = \lfloor \log m \rfloor$, where $\lfloor \cdot \rfloor$ returns the integer part of the argument. Then the system $(a_1, \cdots, a_h) \prod_{i \in 1..m} P_i^1$ where

$$\begin{aligned} P_i^\ell &= \overline{a_\ell} \mid [a_\ell, a_\ell].P_i^{\ell+1} \qquad (1 \le \ell \le h-1) \\ &= \\ P_i^h &= \overline{a_h} \mid [a_h, a_h].\overline{out_i} \end{aligned}$$

solves the leader election in a symmetric network. (When $m = 3$, $P_i = \overline{a_1} \mid [a_1, a_1].\overline{out_i}$, with $i = 1, 2, 3$.) In the sense that exactly one $\overline{out_k}$ will be performed.

A precise account of our work follows. We extend CCS with a more powerful input prefix $[a_1, \cdots, a_n].P$ that gets simultaneously the outputs on $a_1, \cdots, a_n$ and transits to P. We define a family of process calculi, called CCS^n, that retains input prefixes up-to n names. In order to demonstrate the presence of an expressiveness gap between CCS^n and CCS^{n-1}, we consider a well-known problem of resource condivision: the dining philosophers problem. In facts, we consider a variant of it – the dining philosophers problem in the hypercube – where the philosophers sit at the vertices of an hypercube of dimension n, forks are at the angle, and a philosopher can grab forks at its own angles only. We demonstrate that the problem may be solved in CCS^n but not in CCS^{n-1} if the network of philosophers is symmetric and the solution has no divergent behaviour (detailed later).

Then we compare the expressiveness of joint inputs with another well-known expressive operation of CCS, the (guarded) choice. We show that input-guarded choice $\sum_{i \in I} a_i.P_i$ may be easily encoded already in CCS^2. However our solution for mixed-guarded choice $\sum_{i \in I} \alpha_i.P_i$, where α_i may be either input or output, is not satisfactory. In particular, our solution works when two mixed choices never interact. This problem reveals a limitation of CCS^n (as well as in CCS-like calculi and join-like calculi): in synchronizations the flow of information is in one direction only. If we admitted a more liberal synchronization, such as $[\alpha, \beta].P$, where α and β may be either inputs or outputs then there is a simple encoding of mixed choice:

$$(\ell)(\prod_{i \in I} [\alpha_i, \ell].[![P_i]!] \mid \overline{\ell}) .$$

Related Works

The question about the expressive power of synchronization mechanisms dates back (at least) to the eighties when Francez and Rodeh proposed a distributed, deterministic solution to the dining philosopher problem in CSP [27] and Lehmann and Rabin demonstrated that such a solution does not exist in a language with a synchronization *à la* CCS [36]. After these results, our problem slept for about two decades, till a contribution by Nestmann on the expressive power of joint inputs in pi calculus [43]. Nestmann demonstrated that it is possible to encode the pi calculus with mixed choice into a pi calculus with joint input; however he only conjectured the absence of an encoding from pi-calculus with joint inputs into one with 2-ary joins. Other researches have obtained results similar to our one for the so-called *polyadic synchronizations*, which are interactions confined to two processes (and not to n) but using structured channel names [12, 3].

Apart these researches in process algebras, there are close results in biologically inspired calculi. Actually, the need of implementing biologically inspired calculi in pi calculus has been our first motivation for studying the expressive power of synchronizations. In Chapter 3 we demonstrated that it is not possible to encode the κ-calculus into one admitting at most two reactants – the nanoκ calculus.

Structure of the chapter

In Section 5.2 we define the calculi CCS^n. In Section 5.3 we analyze the expressive power of CCS^n with respect to CCS^{n-1} by studying the descriptions of the dining philosopher problem in the n-hypercube. In Section 5.4 we compare the expressive power of guarded choice and multi-synchronizations. In Section 5.5 we extend CCS into CCS^{n+} with a more liberal multi-synchronizations where input and output can be mixed and show there is an expressiveness gap between CCS^{2+} and CCS^{3+} while CCS^{n+} is encodable into CCS^{3+}.

5.2 CCS with joint inputs

The syntax of CCS^n, called n-join CCS, uses a countable set of *names* $\mathcal{N}$, ranged over by $a, b, c, \cdots$, a countable set of *co-names* $\overline{\mathcal{N}}$, ranged over by $\overline{a}, \overline{b}, \overline{c}, \cdots$, and a countable set of *variables* $\mathcal{V}$, ranged over by $x, y, z, \cdots$.

The syntax of CCS^n is defined by the following grammar:

$$\begin{array}{lcll} P & ::= & \text{”} & \textit{inaction} \\ & \mid & \overline{a}.P & \textit{output} \\ & \mid & [a_1, \cdots, a_m].P & \textit{input}\ (1 \leq m \leq n) \\ & \mid & (a)P & \textit{restriction} \\ & \mid & P \mid P & \textit{parallel} \\ & \mid & x & \textit{variable} \\ & \mid & \texttt{rec}\, x.\, P & \textit{recursion} \end{array}$$

The process ″ defines the terminated process; $\overline{a}.P$ defines a process that sends a message on a and continues as P; the behavior $[a_1, \cdots, a_m].P$ defines a process that receives simultaneously messages on $a_1, \cdots, a_m$ and continues as P. The term $a_1, \cdots, a_m$ represents a multiset; so every permutation of its represents the same multiset (see the structural congruence below). Tailings ″ will be omitted. The parallel allows processes to interact. We often abbreviate the parallel composition of P_i for $i \in I$, where I is a finite set, with $\prod_{i \in I} P_i$. The restriction $(a)P$ limits the scope of a to P; the name a is said to be *bound* in $(a)P$. This is the only binding operator of names in CCS^n. We write $(\widetilde{a})P$ for $(a_1) \cdots (a_n)P$, $n \geq 0$. The *free names* in P, denoted $\texttt{en}(P)$, are the names and co-names in P with a non-bound occurrence. The term $\texttt{rec}\, x.P$ defines a recursive process: a (free) occurrence of the variable x in P stands for the whole $\texttt{rec}\, x.P$. We assume that variables are always bound in processes.

The calculus CCS^1 is Milner's CCS without relabelling and choice [38]. We also notice that inputs in CCS^n have *at most n* messages. One might be stricter on this point, by admitting inputs of CCS^n with *exactly n* messages. However this is an unnecessary constraint, since it is easy to encode inputs with less than n elements into a calculus using only n-multi-synchronizations. For example $[a_1, \cdots, a_{n-1}].P$ may be encoded as $(\ell)(\overline{\ell} \mid [a_1, \cdots, a_{n-1}, \ell].P)$, with ℓ fresh. Said otherwise, CCS^{n-1} is (easily) encodable in CCS^n.

Next we define the *labelled semantics* of CCS^n. There is a subtle issue regarding restriction that deserves to be discussed in advance. Consider the process $(a)([a, b].P \mid \overline{a}.Q) \mid \overline{b}.R$. This process has one τ transition into $(a)(P \mid Q) \mid R$ however we are not able to derive this transition by means of rules like

$$\frac{P_1 \xrightarrow{a,b} P'_1 \qquad P_2 \xrightarrow{\overline{a}} P'_2 \qquad P_3 \xrightarrow{\overline{a}} P'_3}{P_1 \mid P_2 \mid P_3 \xrightarrow{\tau} P'_1 \mid P'_2 \mid P'_3}$$

because of the restriction that encloses two processes only. To overcome this problem we define rules that collect the synchronizing processes *inside* the proof tree, rather than in a unique rule.

We use $\mu, \mu', \cdots$ to range over either sequences of co-names or sequences of names or the special symbol τ. The predicate $a \in \mu$ is true if either a or $\overline{a}$ occurs in the sequence μ, otherwise it is false. Let $a_1, \cdots, a_m \setminus a$ be the function returning

- τ, if $a_1, \cdots, a_m = a$,
- $a_2, \cdots, a_m$, if $a_1 = a$ and $m \geq 2$,
- $a_1, (a_2, \cdots, a_m \setminus a)$, if $a_1 \neq a$ and $a \in a_2, \cdots, a_m$.

The function $a_1, \cdots, a_m \setminus a$ is partial: it is not defined if $a \notin a_1, \cdots, a_m$. This function will be also applied to arguments that are both sequences: $a_1, \cdots, a_m \setminus b_1, \cdots, b_\ell = (\cdots(a_1, \cdots, a_m \setminus b_1) \setminus \cdots) \setminus b_\ell$. Similarly we extend $\in$ to sequences and write $a_1, \cdots, a_m \in b_1, \cdots, b_\ell$ if, for every i, $a_i \in (b_1, \cdots, b_\ell \setminus a_1, \cdots, a_{i-1})$ (this is multiset containment).

The operational semantics of CCS^n is defined by the following rules (plus the symmetric ones for $|$).

$$\overline{a}.P \xrightarrow{\overline{a}} P \qquad [a_1, \cdots, a_m].P \xrightarrow{a_1, \cdots, a_m} P$$

$$\frac{P \xrightarrow{\mu} Q \quad a \notin \mu}{(a)P \xrightarrow{\mu} (a)Q} \qquad \frac{P\{\texttt{rec}\, x.P/_x\} \xrightarrow{\mu} Q}{\texttt{rec}\, x.P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \frac{P \xrightarrow{\overline{a_1}, \cdots, \overline{a_m}} P' \quad Q \xrightarrow{\overline{b_1}, \cdots, \overline{b_n}} Q'}{P \mid Q \xrightarrow{\overline{a_1}, \cdots, \overline{a_{m+n}}} P' \mid Q'}$$

$$\frac{P \xrightarrow{a_1, \cdots, a_m} Q \quad P' \xrightarrow{\overline{a_1}, \cdots, \overline{a_\ell}} Q' \quad a_1, \cdots, a_\ell \in a_1, \cdots, a_m}{P \mid P' \xrightarrow{a_1, \cdots, a_m \setminus a_1, \cdots, a_\ell} Q \mid Q'}$$

All the rules are standard except synchronization that is defined by the last one. Processes performing outputs are collected by the process performing the input. Every time the label is updated according to the output process that has been recruited. When a simple input label like a remains and the process recruits one emiting $\overline{a}$, a τ transition is performed, and no other process may be recruited. Similarly when the input label is $a_1, \cdots, a_\ell$ and the recruited process has ℓ subprocesses performing $\overline{a_i}$, $1 \leq i \leq \ell$, respectively. For example, $(a)([a,b].P \mid \overline{a}.Q) \mid \overline{b}.R \xrightarrow{\tau} (a)(P \mid Q) \mid R$ because $(a)([a,b].P \mid \overline{a}.Q) \xrightarrow{b} (a)(P \mid Q)$ and $\overline{b}.R \xrightarrow{\overline{b}} R$. Also $(a)([a,b].P \mid (\overline{a}.Q \mid \overline{b}.R) \xrightarrow{\tau} (a)(P \mid Q) \mid R$ because $[a,b].P \xrightarrow{a,b} P$ and $\overline{a}.Q \mid \overline{b}.R \xrightarrow{\overline{a},\overline{b}} Q \mid R$ (the synchronization between $[a,b].P$ and $\overline{a}.Q \mid \overline{b}.R$ should have not been possible without labels $\overline{a}, \overline{b}$).

In the following we abbreviate $P \xrightarrow{\tau}{}^{*} \xrightarrow{\mu} \xrightarrow{\tau}{}^{*} Q$ with $P \overset{\mu}{\Longrightarrow} Q$.

In process calculi it is usual to equate processes that differ for alpha equivalence, the abelian monoid law of $\mid$ (associativity, commutativity and $''$ as identity), and the scope laws

$$(a)'' \equiv'', \quad (a)(b)P \equiv (b)(a)P,$$
$$P \mid (a)Q \equiv (a)(P \mid Q), \quad \textit{if } z \notin \mathsf{en}(P)$$

In addition, in CCS^n, we also equate

$$[a_1, \cdots, a_m].P \equiv [a_{i_1}, \cdots, a_{i_m}].P$$

when $i_1, \cdots, i_m$ is a permutation of $1, \cdots, m$.

Let *structural congruence*, noted $\equiv$, be the least congruence containing the above laws. In alternative to the labelled semantics, one may supply CCS^n with a *reduction semantics* consisting of the rule

$$\prod_{i \in 1..n} \overline{a_i}.P_i \mid [a_1, \cdots, a_m].P \;\longrightarrow\; \prod_{i \in 1..n} P_i \mid P$$

We have adhered to a labelled approach in order to ease our arguments in the technical part (see Theorem 5.2).

Let ρ be a *renaming*, that is a map $\mathcal{N} \to \mathcal{N}$, and let $\rho(\overline{a}) = \overline{\rho(a)}$. Then $\rho(P)$ is the process where ρ has been applied homomorphically to every CCS^n operator. Renamings are ranged over by $\rho, \sigma, \cdots$.

5.3 The dining philosophers problem in the hypercube

The usual description of Dijkstra's dining philosophers problem is the following [25]. There are m philosophers sitting around a table with exactly one fork in between them. Philosophers go indefinitely through the following cycle: thinking, trying to eat, and eating. In order to eat, a philosopher needs both the forks on his left and right sides; when the two forks at his sides are free, the philosopher grabs them – one after the other –, eats, and releases them (and starts thinking). The difficulty of the problem is when every philosopher grabs the fork at his left. Then, to escape the deadlock, someone has to release the fork. This fact, assuming that philosophers are identical, may get back to the initial state, thus yielding a cycle.

The dining philosopher problem has been studied since long time. Solutions have been proposed that totally order either forks or philosophers [16] or that use powerful operators, such as the CSP synchronization [27], or probabilistic algorithms [36]. It is worth to observe that the presence of, even simple, solutions of the problem do not invalidate Lehmann and Rabin's theorem in [36] (*there is no deterministic, deadlock-free, truly distributed and symmetric solution to the dining philosopher problem*). But, rather, as they already pointed out for the CSP solution in [27], that there is no truly distributed *implementation* of the synchronization operators we are studying.

The generalization of the dining philosopher problem is when the philosophers sit at the vertices of an *hypercube* and forks are at the edges. Thus every philosopher has n neighbour philosophers and n adjacent forks. In this general problem, a philosopher eats if he grabs all the n adjacent forks. The Figure 5.1 illustrates the cube and the forks (the 3-hypercube) where philosophers, represented by 3 bits, sit at the vertices and forks label edges. In the case of the n-hypercube, the philosophers are represented by n bits – they are 2^n – and the forks by unordered pairs of neighbour philosophers representations – they are $n \times 2^{n-1}$. It is worth to notice that two philosophers are neighbours if their representations differ for exactly one bit.

Few preliminary notions follow. We use $b, b', \cdots$ to range over $\{0,1\}$ and $\widetilde{b}, \widetilde{b'}, \cdots$ to range over $\{0,1\}^n$ (n bits). Let $0 \oplus 0 = 0$, $0 \oplus 1 = 1 \oplus 0 = 1$ and $1 \oplus 1 = 0$

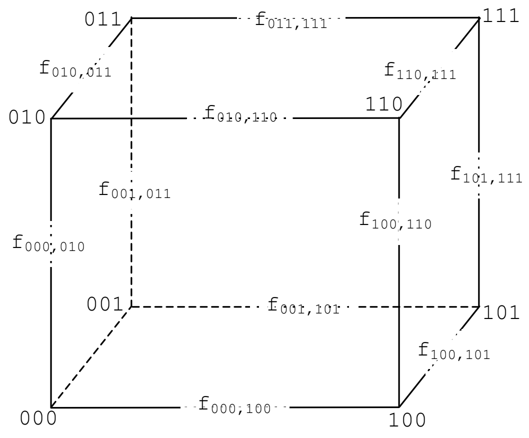


Figure 5.1: The cube (3-hypercube) and the forks

and let $b_1 \cdots b_n \oplus 1$, the *set of neighbours* of $b_1 \cdots b_n$, be $\{(b_1 \oplus 1)b_2 \cdots b_n,\ b_1(b_2 \oplus 1)b_3 \cdots b_n, \cdots, b_1b_2b_3 \cdots (b_n{\oplus}1)\}$. Let also F^n, the forks of the n-hypercube, be $\{f_{\{\widetilde{b},\widetilde{b'}\}} \mid \widetilde{b} \in \{0,1\}^n$ and $\widetilde{b'} \in \widetilde{b} \oplus 1\}$ and $F_{\widetilde{b}}$, the forks that are adjacent to the philosopher $\widetilde{b}$, be $\{f_{\{\widetilde{b},\widetilde{b'}\}} \mid \widetilde{b'} \in \widetilde{b} \oplus 1\}$, For example, when $n = 3$, the set F_{000} is $\{f_{\{000,001\}}, f_{\{000,010\}}, f_{\{000,100\}}\}$; the set notation as index of a fork allows us to equate the forks $f_{\{000,001\}}$ and $f_{\{001,000\}}$.

Dining philosophers in a hypercube network. A *philosopher network* **N** of size n is a $(2^n + 1)$-tuple $\langle P_{0\cdots 0}, \cdots, P_{1\cdots 1}, \prod_{f \in F \subseteq F^n} \overline{f} \rangle$ where $\mathsf{en}(P_{\widetilde{b}}) = F_{\widetilde{b}} \cup \overline{F_{\widetilde{b}}} \cup \{\overline{eat_{\widetilde{b}}}\}$ and the continuations of co-names $\overline{f} \in F_{\widetilde{b}}$ in $P_{\widetilde{b}}$ are always empty. The above network **N** is meant to represent the process

$$(F^n)\Big(\prod_{\widetilde{b} \in \{0,1\}^n} P_{\widetilde{b}} \mid \prod_{f \in F \subseteq F^n} \overline{f}\Big)$$

where $P_{\widetilde{b}}$ will be called the *philosopher at vertex $\widetilde{b}$ of the n-hypercube*. **N** explicitly represent (i) the distribution in the system of the various components of its, (ii) the fact that forks are “passive” processes that may be accessed only by adjacent philosophers, and (iii) that philosophers do not communicate directly.

We also impose another (natural) constraint: that philosophers have all identical code. This is formalized by assuming that there are configurations of the network where the codes of philosophers are equal up-to *bijective renamings*. (The notion of philosopher network describes one of the possible reachable configurations where, for example, philosophers may be in different states.) In particular, in the initial configuration, all philosophers are in the same state and no-one grabs any fork. We say that a philosopher network is *symmetric* if philosophers' codes are identical up-to bijective renamings.

The issue of identical protocols. It is well-known that there are philosopher networks that never manifest a deadlock even if philosophers have identical codes. The reason is that philosophers might use different protocols for grabbing forks. For example, consider the square. Assume that philosophers at even positions grab their right fork and philosophers at odd position grab their left fork. While philosophers are all identical up-to renamings, the competition between those at `00` and `01` and between those at `11` and `10` already allow the progress of only two philosophers. The two "winning" philosophers either will progress grabbing the remaining forks or will compete on a free fork. In any case, the overall system progresses.

A philosopher network **N** of size n is *deadlock-free* if every maximal computation has infinitely many actions in the set $\{\overline{eat}_{\widetilde{b}} \mid \widetilde{b} \in \{0,1\}^n\}$. (This is less demanding than *livelock freedom* where every philosopher is guaranteed to eventually eat.) It is possible to define a deadlock-free philosopher network of size n in CCS^n. The solution is $(F^n)(\prod_{\widetilde{b}\in\{0,1\}^n} Ph_{\widetilde{b}} \mid \prod_{f\in F^n} \overline{f})$, where

$$Ph_{\widetilde{b}} = \mathtt{rec}\, x.[F_{\widetilde{b}}].\overline{eat}_{\widetilde{b}}.(x \mid \prod_{f\in F_{\widetilde{b}}} \overline{f})$$

(we encourage the reader to write the case $n = 3$). We also notice that, considering $Ph_{\widetilde{0}}$, *every* philosopher network of size n containing this process is deadlock-free.

An *Hamiltonian cycle* in the n-hypercube is a path traversing all the vertices of the hypercube without repetition of edges. The following result is due to Alspach, Bermond and Sotteau.

Theorem 5.1 ([5]) *In the n-hypercube there exist $\lfloor n/2 \rfloor$ edge-disjoint Hamiltonian cycles.*

Definition 5.1 *An m-*edge assignment *in the n-hypercube ($m \leq n$) is a map χ from nodes to edges such that $\chi(\widetilde{b}) \subseteq F_{\widetilde{b}}$ and*

- *$\chi(\widetilde{b})$ is either empty or has cardinality m;*

- *for every pairs of neighbours $\widetilde{b}, \widetilde{b'}$, $\chi(\widetilde{b}) \cap \chi(\widetilde{b'}) = \emptyset$ (an edge is assigned to at most one node).*

An m-edge assignment χ in the n-hypercube

- *is* maximal *if, for every m-edge assignment χ' such that, for every $\widetilde{b}$, $\chi(\widetilde{b}) \subseteq \chi'(\widetilde{b})$, then $\chi = \chi'$ (no further m-edge assignment can be done);*

- *(when $m < n$) is* saturated *if it is maximal and, for every $\chi(\widetilde{b}) \neq \emptyset$, there exist $\widetilde{b'} \in \widetilde{b} \oplus 1$ such that $F_{\widetilde{b}} \setminus \chi(\widetilde{b}) \cap F_{\widetilde{b'}} \neq \emptyset$ (some missing edge has been assigned to an adjacent node).*

Let S be a set of edges, we define $S^{\uparrow \widetilde{b}}$ as the set $\{\widetilde{bb'} \leftrightarrow \widetilde{bb''} \mid \widetilde{b'} \leftrightarrow \widetilde{b''} \in S\}$. For example $\{00 \leftrightarrow 01, 01 \leftrightarrow 11\}^{\uparrow 1}$ is $\{100 \leftrightarrow 101, 101 \leftrightarrow 111\}$, namely a set of edges in the cube.

Lemma 5.1 *There exists a maximal n-edge assignment in the n-hypercube.*

Proof. The cases of the square and the cube are left as an exercise. We demonstrate that,

(5.1)(a) for every $n \geq 4$, there are two *canonical* maximal assignments in the n-hypercube, called $\chi^n_{[0]}$ and $\chi^n_{[3]}$, such that, for every $\widetilde{b}$:

$$\chi^n_{[0]}(\widetilde{b}) = \emptyset \qquad \text{if and only if} \qquad \chi^n_{[3]}(\widetilde{b}) \neq \emptyset \ .$$

Let $\chi^4_{[0]}$ and $\chi^4_{[3]}$ be

$\chi^4_{[0]}$ assigns all the adjacent edges to 0000, 0110, 1011, and 1101 – see Figure 5.2;

$\chi^4_{[3]}$ assigns all the adjacent edges to 0011, 0101, 1000, and 1110 – see Figure 5.3.

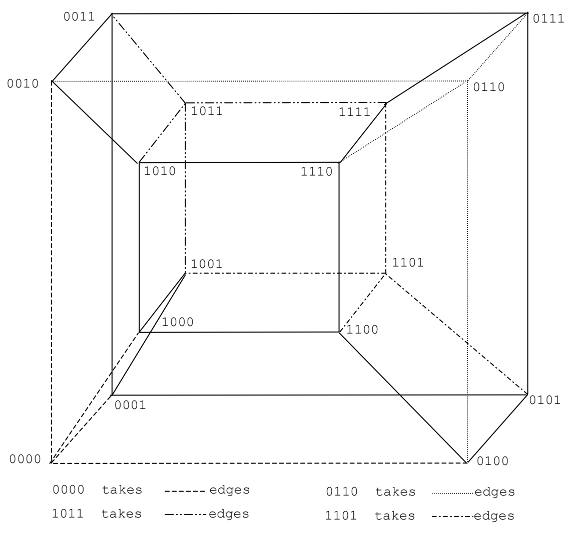


Figure 5.2: The edge assignment $\chi^4_{[0]}$

It is easy to verify that both $\chi^4_{[0]}$ and $\chi^4_{[3]}$ satisfy (5.1)(a). Assuming $\chi^n_{[0]}$ and $\chi^n_{[3]}$ satisfy (5.1)(a), let $\chi^{n+1}_{[0]}$ and $\chi^{n+1}_{[3]}$ be the following assignments:

$$\chi^{n+1}_{[0]}(\widetilde{b}) = \begin{cases} \chi^n_{[0]}(\widetilde{b'})^{\uparrow 0} \cup \{0\widetilde{b'} \leftrightarrow 1\widetilde{b'}\} & \text{if } \widetilde{b} = 0\widetilde{b'} \text{ and } \chi^n_{[0]}(\widetilde{b'}) \neq \emptyset \\ \emptyset & \text{if } \widetilde{b} = 0\widetilde{b'} \text{ and } \chi^n_{[0]}(\widetilde{b'}) = \emptyset \\ \chi^n_{[3]}(\widetilde{b'})^{\uparrow 1} \cup \{0\widetilde{b'} \leftrightarrow 1\widetilde{b'}\} & \text{if } \widetilde{b} = 1\widetilde{b'} \text{ and } \chi^n_{[3]}(\widetilde{b'}) \neq \emptyset \\ \emptyset & \text{if } \widetilde{b} = 1\widetilde{b'} \text{ and } \chi^n_{[3]}(\widetilde{b'}) = \emptyset \end{cases}$$

$$\chi^{n+1}_{[3]}(\widetilde{b}) = \begin{cases} \chi^n_{[3]}(\widetilde{b'})^{\uparrow 0} \cup \{0\widetilde{b'} \leftrightarrow 1\widetilde{b'}\} & \text{if } \widetilde{b} = 0\widetilde{b'} \text{ and } \chi^n_{[3]}(\widetilde{b'}) \neq \emptyset \\ \emptyset & \text{if } \widetilde{b} = 0\widetilde{b'} \text{ and } \chi^n_{[3]}(\widetilde{b'}) = \emptyset \\ \chi^n_{[0]}(\widetilde{b'})^{\uparrow 1} \cup \{0\widetilde{b'} \leftrightarrow 1\widetilde{b'}\} & \text{if } \widetilde{b} = 1\widetilde{b'} \text{ and } \chi^n_{[0]}(\widetilde{b'}) \neq \emptyset \\ \emptyset & \text{if } \widetilde{b} = 1\widetilde{b'} \text{ and } \chi^n_{[0]}(\widetilde{b'}) = \emptyset \end{cases}$$

The proof that $\chi^{n+1}_{[0]}$ and $\chi^{n+1}_{[3]}$ satisfy the constraints in (5.1)(a) follow directly by induction and by definition. □

Lemma 5.2 *There exists a saturated* $(n-1)$*-edge assignment in the n-hypercube.*

Proof. By cases on n.

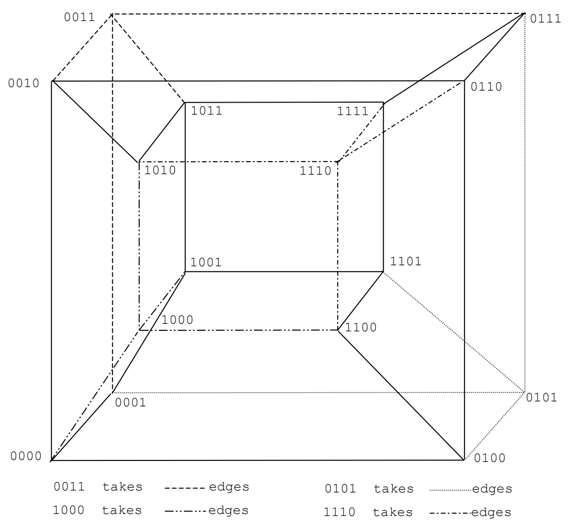


Figure 5.3: The edge assignment $\chi^4_{[3]}$

Case $n = 2$. The 2-hypercube is a square. A maximal 1-edge assignment is obtained by giving to every node its right edge.

Case $n = 3$. The 3-hypercube is a cube, namely the cartesian product of two squares at level 0 and 1, respectively. A maximal 2-edge assignment is obtained by taking the 1-edge assignment of the case $n = 2$ for the square of level 0 – the vertices $000, 001, 010, 011$ – and lifting it to a 2-edge assignment by assigning the edge between the two levels to the corresponding vertices of level 0. Additionally, the assignment allocates $100 \leftrightarrow 110, 100 \leftrightarrow 101$ to 100 and $110 \leftrightarrow 111, 101 \leftrightarrow 111$ to 111.

Case $n = 4$. The 4-hypercube is the cartesian product of two cubes at level 0 and 1, respectively. A maximal 3-edge assignment is obtained by taking the 2-edge assignment of the case $n = 3$ for the cube of level 0 – the vertices $0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111$ – and lifting it to a 3-edge assignment by assigning the forks between the two levels to the corresponding vertices of level 0. Additionally, the assignment allocates the edges $1010 \leftrightarrow 1110, 1010 \leftrightarrow 1011, 1010 \leftrightarrow 1000$ to 1010, $1001 \leftrightarrow 1011, 1001 \leftrightarrow 1000, 1001 \leftrightarrow 1101$ to 1001, $1111 \leftrightarrow 1011, 1111 \leftrightarrow 1110, 1111 \leftrightarrow 1101$ to 1111, $1100 \leftrightarrow 1101, 1100 \leftrightarrow 1110, 1100 \leftrightarrow 1000$ to 1100.

Case $n \geq 5$. The n-hypercube is the cartesian product of two $(n-1)$-hypercubes at level 0 and 1, respectively. Let $\chi^{n-1}_{[0]}$ be the maximal $(n-1)$-edge assignment for the $(n-1)$-

hypercube defined in Proposition 5.1. We define χ, a maximal $(n-1)$-edge assignment for the n-hypercube, as follows:

- $\chi(0\widetilde{b}) = \chi^{n-1}_{[0]}(\widetilde{b})^{\uparrow 0}$;
- whenever $\chi^{n-1}_{[0]}(\widetilde{b}0000) \neq \emptyset$:
  - $\chi(1\widetilde{b}0000) = (\chi^{n-1}_{[0]}(\widetilde{b}0000)^{\uparrow 1} \setminus \{1\widetilde{b}0000 \leftrightarrow 1\widetilde{b}0010\}) \cup \{0\widetilde{b}0000 \leftrightarrow 1\widetilde{b}0000\}$;
  - $\chi(1\widetilde{b}0110) = (\chi^{n-1}_{[0]}(\widetilde{b}0110)^{\uparrow 1} \setminus \{1\widetilde{b}0110 \leftrightarrow 1\widetilde{b}0100\}) \cup \{0\widetilde{b}0110 \leftrightarrow 1\widetilde{b}0110\}$;
  - $\chi(1\widetilde{b}1011) = (\chi^{n-1}_{[0]}(\widetilde{b}1011)^{\uparrow 1} \setminus \{1\widetilde{b}1011 \leftrightarrow 1\widetilde{b}1001\}) \cup \{0\widetilde{b}1011 \leftrightarrow 1\widetilde{b}1011\}$;
  - $\chi(1\widetilde{b}1101) = (\chi^{n-1}_{[0]}(\widetilde{b}1101)^{\uparrow 1} \setminus \{1\widetilde{b}1101 \leftrightarrow 1\widetilde{b}1111\}) \cup \{0\widetilde{b}1101 \leftrightarrow 1\widetilde{b}1101\}$;
  - $\chi(1\widetilde{b}0010) = F_{1\widetilde{b}0010} \setminus \{1\widetilde{b}0010 \leftrightarrow 1\widetilde{b}0110\}$;
  - $\chi(1\widetilde{b}0100) = F_{1\widetilde{b}0100} \setminus \{1\widetilde{b}0100 \leftrightarrow 1\widetilde{b}0000\}$;
  - $\chi(1\widetilde{b}1001) = F_{1\widetilde{b}1001} \setminus \{1\widetilde{b}1001 \leftrightarrow 1\widetilde{b}1101\}$;
  - $\chi(1\widetilde{b}1111) = F_{1\widetilde{b}1111} \setminus \{1\widetilde{b}1111 \leftrightarrow 1\widetilde{b}1011\}$;
- whenever $\chi^{n-1}_{[0]}(\widetilde{b}0011) \neq \emptyset$:
  - $\chi(1\widetilde{b}0011) = (\chi^{n-1}_{[0]}(\widetilde{b}0011)^{\uparrow 1} \setminus \{1\widetilde{b}0011 \leftrightarrow 1\widetilde{b}0111\}) \cup \{0\widetilde{b}0011 \leftrightarrow 1\widetilde{b}0011\}$;
  - $\chi(1\widetilde{b}0101) = (\chi^{n-1}_{[0]}(\widetilde{b}0101)^{\uparrow 1} \setminus \{1\widetilde{b}0001 \leftrightarrow 1\widetilde{b}0101\}) \cup \{0\widetilde{b}0101 \leftrightarrow 1\widetilde{b}0101\}$;
  - $\chi(1\widetilde{b}1000) = (\chi^{n-1}_{[0]}(\widetilde{b}1000)^{\uparrow 1} \setminus \{1\widetilde{b}1000 \leftrightarrow 1\widetilde{b}1100\}) \cup \{0\widetilde{b}1000 \leftrightarrow 1\widetilde{b}1000\}$;
  - $\chi(1\widetilde{b}1110) = (\chi^{n-1}_{[0]}(\widetilde{b}1110)^{\uparrow 1} \setminus \{1\widetilde{b}1110 \leftrightarrow 1\widetilde{b}1010\}) \cup \{0\widetilde{b}1110 \leftrightarrow 1\widetilde{b}1110\}$;
  - $\chi(1\widetilde{b}0001) = F_{1\widetilde{b}0001} \setminus \{1\widetilde{b}0001 \leftrightarrow 1\widetilde{b}0011\}$;
  - $\chi(1\widetilde{b}0111) = F_{1\widetilde{b}0111} \setminus \{1\widetilde{b}0101 \leftrightarrow 1\widetilde{b}0111\}$;
  - $\chi(1\widetilde{b}1010) = F_{1\widetilde{b}1010} \setminus \{1\widetilde{b}1010 \leftrightarrow 1\widetilde{b}1000\}$;
  - $\chi(1\widetilde{b}1100) = F_{1\widetilde{b}1100} \setminus \{1\widetilde{b}1100 \leftrightarrow 1\widetilde{b}1110\}$;
- otherwise: $\chi(1\widetilde{b}) = \emptyset$.

Informally, χ lifts the maximal $(n-1)$ edge assignment $\chi^{n-1}_{[0]}$ to the $(n-1)$-hypercube at level 0, and uses a modified version of $\chi^{n-1}_{[0]}$ for the $(n-1)$-hypercube at level 1. In particular, the edges in between the two levels are assigned to vertices at level 1, whilst the same vertices leave exactly one edge in favour of new vertices that are adjacent to two of them. We leave the reader to verify that either $\chi(\widetilde{b}) = \emptyset$ or $\chi(\widetilde{b})$ is a set of $n-1$ forks. The fact that χ is saturated follows directly by definition. □

Lemma 5.3 *Let* $1 \leq m \leq n-2$*. There exists a saturated m-edge assignment in the n-hypercube.*

Proof. There are two cases: $1 \leq m \leq \lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1 \leq m \leq n-2$.

$1 \leq m \leq \lfloor n/2 \rfloor$: By Theorem 5.1 there are $\lfloor n/2 \rfloor$ Hamiltonian cycles in the n-hypercube. We consider m of them and we fix a direction for every cycle. The m-edge assignment χ associates to every vertex its outgoing m edges. In this case $\chi(\widetilde{b}) \neq \emptyset$, for every $\widetilde{b}$, and it is easy to verify that it is saturated.

$\lfloor n/2 \rfloor + 1 \leq m \leq n-2$: Let H_{m+1} be an $(m+1)$-hypercube. The n-hypercube is

$$\underbrace{H_{m+1} \times \cdots \times H_{m+1}}_{2^{n-(m+1)} times}$$

By Lemma 5.2, there exists a saturated m-edge assignment in H_{m+1} and let it be $\chi^{[m+1]}$. In the following of the proof, let $\widetilde{b}$ range over $\{0,1\}^{n-(m+1)}$. We define an edge assignment χ' of the n hypercube as follows:

$$\chi'(\widetilde{b}\widetilde{b'}) = \chi^{[m+1]}(\widetilde{b'})^{\uparrow \widetilde{b}} .$$

By construction, χ' is such that, for every $\widetilde{b''}$ there exist $\widetilde{b'''} \in \widetilde{b''} \oplus 1$ with $F_{\widetilde{b''}} \cap \chi'(\widetilde{b''}) \neq \emptyset$. However χ' is not maximal because there are vertices $\widetilde{b''}$ with $\chi'(\widetilde{b''}) = \emptyset$ and, yet, m unassigned edges in $F_{\widetilde{b''}}$. These vertices had not enough adjacent unassigned edges in H_{m+1}, but have enough edges in the n-hypercube due to the presence of $n-(m+1)$ further edges. Clearly, it is possible to extend χ' by performing a sequence of edge assignments. When no other edge assignment is possible, one gets a χ that is maximal. It is also saturated because of the corresponding property for χ'. □

In a network philosopher system, a philosopher releasing a fork that he has just grabbed cannot affect the behaviour of other philosophers because the communication through the forks is asynchronous.For example, if the code of a philosopher is $[f_1, f_2, f_3].(\overline{f_2} \mid \overline{f_3} \mid P)$, the other philosophers will not distinguish it from the code $[f_1].P$. It is also possible to iterate the same argument after the first input. That is, in the cube, the code $[f_1, f_2].(\overline{f_2} \mid [f_2].(\overline{f_1} \mid P))$ is undistinguishable from $[f_2].P$. This asynchronicity is formalized by the following auxiliary relation $\rightsquigarrow$, which is the least one satisfying the rules:

$$\frac{\begin{array}{c} P \overset{f_1,\cdots,f_m}{\Longrightarrow} \overset{\overline{g_1}}{\Longrightarrow} \cdots \overset{\overline{g_\ell}}{\Longrightarrow} P' \overset{\overline{g_{\ell+1}}}{\not\Rightarrow} \\ g_1,\cdots,g_{\ell+1} \subseteq f_1,\cdots,f_m \\ h_1,\cdots,h_\kappa = (f_1,\cdots,f_m \setminus g_1,\cdots,g_\ell) \end{array}}{P \overset{h_1,\cdots,h_\kappa}{\rightsquigarrow} P'}$$

$$\frac{\begin{array}{c} P \overset{f_1,\cdots,f_m}{\rightsquigarrow} P' \quad P' \overset{f'_1,\cdots,f'_{m'}}{\Longrightarrow} \overset{\overline{g_1}}{\Longrightarrow} \cdots \overset{\overline{g_\ell}}{\Longrightarrow} P'' \overset{\overline{g_{\ell+1}}}{\not\Rightarrow} \\ g_1,\cdots,g_{\ell+1} \subseteq f_1,\cdots,f_m,f'_1,\cdots,f'_{m'} \\ h_1,\cdots,h_\kappa = (f_1,\cdots,f_m,f'_1,\cdots,f'_{m'} \setminus g_1,\cdots,g_\ell) \end{array}}{P \overset{h_1,\cdots,h_\kappa}{\rightsquigarrow} P''}$$

It is worth to notice that $\rightsquigarrow$ is an abbreviation for *sequences of transitions labelled by forks*. Transitions labelled by *eat* are not considered and are never performed. In facts, the above considerations about asynchrony are false for labels *eat* in a philosopher network. In the second rule of $\rightsquigarrow$, $m + m' \leq n$, otherwise a philosopher would try to grab a fork that he already retains.

Theorem 5.2 *Let P be the CCS^{n-1} code of a philosopher at vertex $\widetilde{b}$ of the n-hypercube that does not retain any fork. There exists a symmetric philosopher network* $\mathbf{N}[P]$ *having P at position $\widetilde{b}$ that is not deadlock-free.*

Proof: Without loss of generality, we assume P be the process at vertex $\widetilde{0}$. Let $P = P_{\widetilde{0}} \overset{f_1^{\widetilde{0}},\cdots,f_m^{\widetilde{0}}}{\rightsquigarrow} P'_{\widetilde{0}} \overset{g_1^{\widetilde{0}},\cdots,g_{m'}^{\widetilde{0}}}{\longrightarrow}$ be the longest computation such that $m + m' = n$, $m < n$ and $m' < n$, and the other input transitions in $P'_{\widetilde{0}}$ have labels with sequences longer than m'. Few remarks are in order:

– $P_{\widetilde{0}}$ cannot output because he does not grab forks;

– there may be several different inputs due to nondeterminism: we are considering one of them;

– by definition of $\rightsquigarrow$, only input moves are possible in $P'_{\widetilde{0}}$.

If there is no finite longest computation as above then either P will block on an internal action or it has a divergent computation. In any case, it is easy to define a philosopher network $\mathbf{N}[P]$ that is not deadlock free. Otherwise, the argument is by cases on m.

$1 \leq m \leq n-2$: By Lemma 5.3, there exists a saturated m-edge assignment χ in the n-hypercube. Let $\{f_1^{\widetilde{0}}, \cdots, f_m^{\widetilde{0}}\} = \chi(\widetilde{0})$ and let $\mathbf{N}[P]$ be the network such that $P_{\widetilde{b}} = \rho_{\widetilde{b}}(P_{\widetilde{0}})$ with $\rho_{\widetilde{b}}(\{f_1^{\widetilde{0}}, \cdots, f_m^{\widetilde{0}}\}) = \chi(\widetilde{b})$. Since χ is saturated, there exists a computation $\mathbf{N}[P] \Longrightarrow \mathbf{N}'$, where the philosopher at $\widetilde{b}$ is either in the state $P'_{\widetilde{b}}$ or in a state reachable from $P_{\widetilde{b}}$ with internal moves. In both cases, because χ is saturated, he cannot perform either the transition $\xrightarrow{g_1^{\widetilde{b}}, \cdots, g_{m'}^{\widetilde{b}}}$ or the transition $\xRightarrow{f_1^{\widetilde{b}}, \cdots, f_m^{\widetilde{b}}}$. So the network is deadlocked.

$m = n-1$: Let χ be the saturated $(n-1)$-fork assignment of Lemma 5.2. We define the renaming $\rho_{\widetilde{b}}$ from $P_{\widetilde{0}}$ to $P_{\widetilde{b}}$ in such a way that $\rho_{\widetilde{b}}(\{f_1^{\widetilde{0}}, \cdots, f_{n-1}^{\widetilde{0}}\}) = \chi(\widetilde{b})$. The proof is similar to the previous case. □

It is worth to observe that Theorem 5.2 also holds when P retains at most $\lceil n/2 \rceil$ forks because, in these cases, it is possible to define a symmetric philosopher network. As a consequence of Theorem 5.2, it is possible to demonstrate that CCS^n is not encodable into CCS^{n-1}, under certain requirements for the notion of encoding. Following Palamidessi [59], let $[![\cdot]!]$ be *uniform* if

– it is *compositional*;

– it is *renaming preserving*, namely for every injective renaming θ on names of P there exists an injective renaming θ' such that $[![(\theta)(P)]!] = (\theta')([![P]!])$

Compositionality ensures that the encoding of a compound process must be expressed in terms of the encoding of its components. However, in a distributed context (as the one of philosophers networks), the requirement of compositionality is usually strengthened by requiring the *homomorphism with respect to "* $|$*"*, namely

$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$$

that is, the encoding must preserve the degree of distribution of the processes (parallel processes remain in parallel) *and* cannot introduce additional processes that might act as coordinators.

The requirement of renaming preserving means that the encoding does not depend on the identity of free (channel) names, which is understandable as long as one wants liberal installations of processes in the network. We assume that the renamings θ and θ' map names into names (the encoding uses a *strict renaming policy* [3]). In addition, as in [45], Section 7, the renaming preserving constraint is strengthened into $\theta(a) = \theta'(a)$, for every *relevant free name*. This strengthening aims at substantiating that external resources must be accessed in the same way by the process and every encoding of its, hence encodings cannot rename them. In the dining philosopher networks, the relevant free name are the forks and names *eat*. Therefore we will assume $\theta(f_{\widetilde{b},\widetilde{b'}}) = \theta'(f_{\widetilde{b},\widetilde{b'}})$ and $\theta(eat_{\widetilde{b}}) = \theta'(eat_{\widetilde{b}})$.

Let $[\![\cdot]\!]$ be *semantically reasonable* if it preserves the relevant observables and the termination properties. Usually, the observables that are relevant in a concurrent scenario are the sequence of interactions with possible parallel contexts. As regards termination, reasonableness constrains the encoding in not introducing divergence.

Corollary 5.1 *There exists no uniform, semantically reasonable encoding of* CCS^n *into* CCS^{n-1}.

Proof: Assume to the contrary that $[\![\cdot]\!]$ is a uniform, semantically reasonable encoding of CCS^n into CCS^{n-1}. Take the philosopher code in CCS^n:

$$\begin{aligned} Ph_{\widetilde{0}} = \;& [f_1, \cdots, f_n].\overline{eat}_{\widetilde{0}}. \\ & (\textstyle\prod_{f \in F_{\widetilde{0}}} \overline{f} \mid \texttt{rec}\, x.[f_1, \cdots, f_n].\overline{eat}_{\widetilde{0}}.(\prod_{f \in F_{\widetilde{0}}} \overline{f} \mid x)) \end{aligned}$$

where $f_1, \cdots, f_n$ are the forks adjacent to the philosopher at $\widetilde{0}$. It is worth to notice that the network is deadlock free *for every possible permutation* of forks in the prefixes $[f_1, \cdots, f_n]$ because, in this case, the philosopher is grabbing all his adjacent forks at a time.

By Theorem 5.2, there exists a family $\rho_{\widetilde{0},\widetilde{b}}$ of bijective renamings such that the process

$$\prod_{\widetilde{b}\in\{0,1\}^n} \rho_{\widetilde{0},\widetilde{b}}([![Ph_{\widetilde{0}}]!])$$

when put in the context $(F^n)([\ ]\ \mid\ \prod_{f\in F^n}\overline{f})$ is not deadlock free.

The renamings $\rho_{\widetilde{0},\widetilde{b}}$ map forks adjacent to $\widetilde{0}$ to forks adjacent to $\widetilde{b}$ and $eat_{\widetilde{0}}$ to $eat_{\widetilde{b}}$. Let $Ph'_{\widetilde{0}} = \overline{eat}_{\widetilde{0}}.(\prod_{f\in F_{\widetilde{0}}}\overline{f} \mid \texttt{rec}\, x.[f_1, \cdots, f_n].\overline{eat}_{\widetilde{0}}.(\prod_{f\in F_{\widetilde{0}}}\overline{f} \mid x))$. Because the encoding is compositional:

$$[![Ph_{\widetilde{0}}]!] = C_{f_1,\cdots,f_n}[[![Ph'_{\widetilde{0}}]!]]$$

where $C_{f_1,\cdots,f_n}[\cdot]$ is a context encoding the joint input $[f_1, \cdots, f_n]$. Hence, since $[![\cdot]!]$ is uniform and $\rho_{\widetilde{0},\widetilde{b}}$ is a bijective renaming:

$$\begin{aligned}
\rho_{\widetilde{0},\widetilde{b}}([![Ph_{\widetilde{0}}]!]) &= \rho_{\widetilde{0},\widetilde{b}}(C_{f_1,\cdots,f_n}[[![Ph'_{\widetilde{0}}]!]])\\
&= C_{\rho_{\widetilde{0},\widetilde{b}}(f_1),\cdots,\rho_{\widetilde{0},\widetilde{b}}(f_n)}[\rho_{\widetilde{0},\widetilde{b}}([![Ph'_{\widetilde{0}}]!])]\\
&= C_{\rho_{\widetilde{0},\widetilde{b}}(f_1),\cdots,\rho_{\widetilde{0},\widetilde{b}}(f_n)}[[![\rho_{\widetilde{0},\widetilde{b}}(Ph'_{\widetilde{0}})]!]]\\
&= [![\rho_{\widetilde{0},\widetilde{b}}(Ph_{\widetilde{0}})]!]
\end{aligned}$$

We conclude by taking the process in CCS^n

$$\prod_{\widetilde{b}\in\{0,1\}^n} \rho_{\widetilde{0},\widetilde{b}}(Ph_{\widetilde{0}})\ .$$

This process, when put in the context $(F^n)([\ ]\ \mid\ \prod_{f\in F^n}\overline{f})$, has a deadlock free behaviour, while its encoding $[![\cdot]!]$ is not, contradicting our assumption. □

We notice that a similar result may be proved for mobile calculi à la pi calculus with joint inputs that are reminiscent of join calculus [26]. One such language is described in [32, 43].

5.4 Choices and joint inputs

As discussed in the previous section, joint inputs are rather expressive. In this section we compare them with another well-known expressive operation of CCS, the (guarded) choice. In the following $[a].P$ is abbreviated into $a.P$.

The *input guarded choice* in CCS, written $\sum_{i\in I} a_i.P_i$, is defined by the transition rule $\sum_{i\in I} a_i.P_i \xrightarrow{a_j} P_j$. There is a straightforward encoding of the input guarded choice in CCS^2:

$$[![\sum_{i\in I} a_i.P_i]!] \;=\; (\ell)(\prod_{i\in I}[a_i,\ell].[![P_i]!] \;\mid\; \overline{\ell}) \tag{5.1}$$

where $\ell \notin \mathtt{en}(\sum_{i\in I} a_i.P_i)$. (The encoding is uniform and semantically reasonable: it is possible to demonstrate that $P \xrightarrow{\alpha} P'$ if and only if $[![P]!] \xrightarrow{\alpha}\equiv [![P']!]$.) Of course, this translation is not original: it have already been used in join calculus to implement input guarded choice.

A choice operation that is more expressive then input guarded choice is the *mixed choice* [44], written $\sum_{i\in I} \alpha_i.P_i$, where $\alpha_i \in \mathcal{N} \cup \overline{\mathcal{N}}$. This choice may be encoded in CCS^2 plus output-guarded choices as follows. Let $I = I' \cup I''$ such that $\{\alpha_i \mid i \in I'\} \subseteq \mathcal{N}$ and $\{\alpha_i \mid i \in I''\} \subseteq \overline{\mathcal{N}}$. Then

$$[![\sum_{i\in I} \alpha_i.P_i]!] \;=\; (\ell)(\prod_{i\in I'}[\alpha_i,\ell].[![P_i]!] \;\mid\; (\overline{\ell} + \sum_{j\in I''} \alpha_j.[![P_j]!]))$$

(the correctness of the encoding may be proved similarly to the case of input-guarded choice). In this case, the output guarded choices seem to be necessary and we are not aware of any uniform and semantically reasonable encoding of mixed choice into CCS^2.

The above remark paves the way for a calculus alternative to CCS^n. In facts, there is a solution of the problem of encoding mixed choice in a choice-free calculus by replacing joint inputs with *joint prefixes*. Let CCS^{2+} be the extension of CCS^2 with prefixes $[\alpha_1,\alpha_2].P$, where $\alpha_i \in \mathcal{N} \cup \overline{\mathcal{N}}$. We will define the semantics of CCS^{2+} in the next section; for the time being we rely on reader's intuition. Then the mixed choice $\sum_{i\in I} \alpha_i.P_i$ may be encoded in a similar way to (5.1):

$$[![\sum_{i\in I} \alpha_i.P_i]!] \;=\; (\ell)(\prod_{i\in I}[\alpha_i,\ell].[![P_i]!] \;\mid\; \overline{\ell})$$

For example, $[![a.P + \overline{b}.Q]!]$ is $(\ell)([a, \ell].[![P]!] \ \mid \ [\overline{b}, \ell].[![Q]!] \ \mid \ \overline{\ell})$. It is worth to notice that in CCS^{2+}, unlike CCS^{2}, it is possible that more than two processes do synchronize. For example the process in CCS with mixed choice

$$(a.P + \overline{b}.Q) \mid (\overline{a}.P' + b.Q')$$

is translated into

$$\begin{array}{l} (\ell)([a, \ell].[![P]!] \mid [\overline{b}, \ell].[![Q]!] \mid \ \overline{\ell}) \\ \mid (\ell')([\overline{a}, \ell'].[![P']!] \mid [b, \ell'].[![Q']!] \mid \ \overline{\ell'}) \end{array}$$

that requires four parallel processes to sort out the right choices. As we will discuss below, in CCS^{2+}, there is no upper bound to the number of processes that synchronize.

5.5 CCS with joint prefixes

Let α, possibly indexed, range over $\mathcal{N} \cup \overline{\mathcal{N}}$ and let $\overline{\overline{a}} = a$. The calculus CCS^{n+}, called CCS with n-joint prefixes, is CCS^{n} where outputs and inputs are replaced by the *n-joint prefix*

$$[\alpha_1, \cdots, \alpha_m].P$$

with $1 \le m \le n$. As for CCS^{n}, the term $\alpha_1, \cdots, \alpha_m$ represents a sequence (now of names and co-names). With an abuse of notation, let μ, η range over sequences $\alpha_1, \cdots, \alpha_n$ and τ; let ν range over sequences $\alpha_1, \cdots, \alpha_n$ or ε (the empty sequence). Let $\overline{\nu} = \overline{\alpha_1}, \cdots, \overline{\alpha_n}$ if $\nu = \alpha_1, \cdots, \alpha_n$, $\overline{\nu} = \varepsilon$ if $\nu = \varepsilon$. Let also $\alpha \in \mu$ if α occurs in μ. When $\alpha \in \mu$, let $\nu \setminus \alpha$ be

- ε if $\nu = \alpha$;
- $\alpha_2, \cdots, \alpha_m$, if $\nu = \alpha, \alpha_2, \cdots, \alpha_m$;
- $\alpha_1, (\alpha_2, \cdots, \alpha_m) \setminus \alpha$, otherwise.

The operations $\in$ and $\setminus$ are extended to sequences of prefixes as follows:

- $\varepsilon \in \mu$ and $\mu \setminus \varepsilon = \mu$;
- $\mu \setminus (\alpha_1, \cdots, \alpha_m) = (\cdots(\mu \setminus \alpha_1) \setminus \cdots \setminus \alpha_m)$;

– $\alpha_1, \cdots, \alpha_m \in \mu$ if, for every i, $\alpha_i \in (\mu \setminus (\alpha_1, \cdots, \alpha_{i-1}))$.

The operational semantics of CCS^{n+} is defined by the following rules (plus the symmetric ones for | and where we are letting $\varepsilon, \varepsilon = \tau$):

$$[\alpha_1, \cdots, \alpha_n].P \overset{\alpha_1, \cdots, \alpha_n}{\longrightarrow} P$$

$$\frac{P \xrightarrow{\mu} Q \quad a \notin \mu}{(a)P \xrightarrow{\mu} (a)Q} \qquad \frac{P\{\mathtt{rec}\,x.P/_x\} \xrightarrow{\mu} Q}{\mathtt{rec}\,x.P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} Q}{P \mid P' \xrightarrow{\mu} Q \mid P'} \qquad \frac{\begin{array}{c} P \xrightarrow{\mu} Q \qquad P' \xrightarrow{\eta} Q' \\ \mu \neq \tau \neq \eta \quad \nu \in \mu \quad \overline{\nu} \in \eta \end{array}}{P \mid P' \overset{\mu \setminus \nu, \eta \setminus \overline{\nu}}{\longrightarrow} Q \mid Q'}$$

The main difference with the transition relation of CCS^n is that there is no collector of the synchronizing processes – in CCS^n this role was played by the input process – but they aggregate two by two in a more symmetric way. This pairwise aggregation may not cause any synchronization, which is the case when $\nu = \varepsilon$ – in a similar way to the aggregation of outputs in CCS^n. On the contrary, the sychronization is complete when the two processes in parallel show up matching sequences. For example, the process $(a)([\overline{a}, b].P \mid [a, \overline{c}].Q) \mid [\overline{b}, c].R$ has one τ transition into $(a)(P \mid Q) \mid R$ that follows by collecting first $[\overline{a}, b].P$ and $[a, \overline{c}].Q$ and then $[\overline{b}, c].R$. The collection of the first two processes produces a "residual" label $b, \overline{c}$ that matches with the label of the third process. No other process needs to be collected because the result of the match is ε, ε, which represents a τ move.

Proposition 5.1 *Let $n \geq 3$. There exists a uniform, semantically reasonable encoding of CCS^{n+} into CCS^{3+}.*

Proof: The encoding $[\![\cdot]\!]$ is homomorphic with respect to every operation except the prefix $[\alpha_1, \cdots, \alpha_n].P$ whose definition is:

$$
\begin{array}{rl}
[\![[\alpha_1, \cdots, \alpha_n].P]\!] = (\ell_1, \cdots, \ell_n)(& [\ell_1, \alpha_1, \overline{\ell_2}].'' \\
& \mid [\ell_2, \alpha_2, \overline{\ell_3}].'' \\
& \ldots \\
& \mid [\ell_n, \alpha_n, \overline{\ell_1}].[\![P]\!] \;)
\end{array}
$$

It is easy to prove that $P \xrightarrow{\mu} P'$ if and only if $[\![P]\!] \xrightarrow{\mu}\equiv [\![P']\!]$, where $\equiv$ is the same of the one defined for CCS^n. □

However there is an expressiveness gap between two and three that may be disclosed by studying the dining philosopher problem in the cube. The point is that, in CCS^{2+} it is possible to synchronize as many processes as needed *but* with the overall effect of exposing at most two labels. This is too restrictive when resources must be grabbed at once, as in the case of the dining philosophers. We omit the proof because it is similar to the Corollary 5.1.

Proposition 5.2 *There exists no uniform, semantically reasonable encoding of* CCS^{3+} *into* CCS^{2+}.

The relationship between CCS^{2+} and the hierarchy CCS^n remains an open issue. For example, it is possible to encode the mixed choice into CCS^{2+}. Let $I = I' \cup I''$ such that $\{\alpha_i \mid i \in I'\} \subseteq \mathcal{N}$ and $\{\alpha_i \mid i \in I''\} \subseteq \overline{\mathcal{N}}$. Then

$$
\begin{array}{rcl}
[\![\sum_{i \in I} \alpha_i.P_i]\!] & = & (\ell, \ell', \ell'')(\;\; \prod_{i \in I'} [\alpha_i, \ell].[\![P_i]\!] \\
& & \qquad\qquad \mid \; \prod_{i \in I'} [\overline{\ell''}, \alpha_j].[\![P_j]\!] \\
& & \qquad\qquad \mid [\overline{\ell}, \ell'].'' \mid [\ell', \ell''].'' \mid \overline{\ell'} \\
& &)
\end{array}
$$

On the contrary, we have not been able to define a uniform, semantically reasonable encoding of mixed choice in CCS^n.

Chapter 6

Conclusions

Much work still needs to be done in the conjunction process between Systems Biology and Computer Science. So far researchers from Concurrency Theory field have provided many formalisms for the modeling and the analysis of biological systems. Most of them are bio-inspired and tailored to model a specific aspect of biological systems. In this thesis we have focused on two of the most interesting (family of) formalisms and tried to enrich their formal study by providing various expressiveness results.

We have established a bridge between the MDB calculus, whose primitives are inspired by membrane fusion and fission processes and Simple SA systems, whose operations are purely based on communication of objects and closely related to the biologic trans-membrane communication by coupling chemicals. We concretely provide an encoding of Simple SA into MDB. An example of a translation of a simplified sodium–potassium pump model is given in order to explain how the encoding works. Since the MDB calculus is not Turing equivalent our encoding entails Simple SA as a lower bound for the expressive power of MDB.

We addressed the issue of *local-implementation* between κ and nanoκ. We give a positive answer to the local-implementation of κ in nanoκ that is divergent and we show the nonexistence of deterministic solutions retaining "reasonable" properties. The main negative consequence of this last result is the impossibility of implementing a stochastic version of κ in nanoκ (or pi calculus) by preserving the distribution of rates (see [17]).

We investigated the leader election problem within the κ family and introduced a new sub family called $\texttt{ps}^{\texttt{n}}$ in order to characterize the primitives required to solve the problem by the various calculi. Specifically we attempted to compare the calculi by testing their capability in solving the leader election problem in two different scenarios: a fully connected symmetric network and a ring symmetric network. The fully connected network turned out to be a weak comparison ground since every calculi was able to solve the problem in this scenario. The only interesting point is that we provide a unique solving algorithm for κ, $\texttt{nano}\kappa$ and the whole $\texttt{ps}^{\texttt{n}}$ family. Further investigation within the ring scenario pointed out that the *synchronization degree*, i.e. the number of elements we allow to communicate simultaneously, has an important role in this context. We show that the $\texttt{ps}^{\texttt{n}}$ family, where n stands for the maximum number of molecules allowed to react simultaneously, is unable to solve the leader election in general but interestingly any given calculus $\texttt{ps}^{\texttt{m}}$ can solve the leader election when the ring network size is less than $2m - 1$. This is due to the fact that in $\texttt{ps}^{\texttt{m}}$ we can synchronize m elements at a time. A corollary of these result is that the $\texttt{ps}^{\texttt{n}}$ family retains a totally ordered hierarchy wrt the expressive power: $\texttt{ps}^{\texttt{m+1}}$ is strictly more powerful than $\texttt{ps}^{\texttt{m}}$. We then show how to extend the less powerful calculus of the $\texttt{ps}^{\texttt{n}}$ family, i.e. $\texttt{ps}^2$, in order to solve the problem in the ring network. One such extension is obtained adding the bond-flipping primitive of $\texttt{nano}\kappa$, and another by allowing rules with synchronization degree equal to three and bonds creation with no constraint.[1] The first extension is a sub calculus for $\texttt{nano}\kappa$ and the latter for κ while the converse is not true. It is worth to notice that the algorithm solving the problem for the first extension does not depend on the network size while all the other considered in this chapter do.

Mostly inspired by the synchronization mechanisms of the κ family, we investigated the power of *multi-synchronizations* from a purely theoretical point of view within the framework of a well known calculi such as CCS. We demonstrated that a uniform, fully distributed translation of Milner's CCS with multi input synchronizations of $n + 1$ processes into CCS with multi input synchronizations of n processes that retains a "reasonable" semantics does not exist. We then extended our study to CCS with a more powerful multi-synchronization mechanism allowing a process to perform both inputs and outputs

[1]As detailed in Chapter 4 ps has some restrictions on rules.

at the same time. We show that synchronizations containing more than three input/output items are encodable in those with three items, while there is an expressiveness gap between three and two. The multi-synchronization seems a rather powerful primitive and an interesting goal for the near future will be to investigate how it is affected by asynchronous scenarios.

Several expressiveness results for bio-inspired languages with a strong focus on the κ family have been illustrated throughout this thesis and, along with some detour in Theoretical Computer Science, hopefully our initial goal has been satisfied.

References

[1] The p systems web page. `http://psystems.disco.unimib.it/`.

[2] *21st Annual Symposium on Foundations of Computer Science, 13-15 October 1980, Syracuse, New York, USA*. IEEE, 1980.

[3] *Towards a Unified Approach to Encodability and Separation Results for Process Calculi*, volume 5201 of *Lecture Notes in Computer Science*. Springer, 2008.

[4] Luca Aceto and Anna Ingólfsdóttir, editors. *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, volume 3921 of *Lecture Notes in Computer Science*. Springer, 2006.

[5] B. Alspach, J.-C. Bermond, and D. Sotteau. Decomposition into cycles I: Hamilton decompositions. In *Proceedings of 1987 Cycles and Rays Colloquium, Montréal*, pages 9–18. NATO ASI Ser. C, Kluwer Academic Publishers, Dordrech, 1990.

[6] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC*, pages 82–93, 1980.

[7] Luc Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Inf.*, 25(2):179–201, 1988.

[8] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[9] Nadia Busi. On the computational power of the mate/bud/drip brane calculus: Interleaving vs. maximal parallelism. In *Pre-Proc. of Sixth International Workshop on Membrane Computing (WMC6), Vienna, June 18-21, 2005*, pages 235–252, 2005.

[10] Nadia Busi and Roberto Gorrieri. On the computational power of brane calculi. *CMSB 2005*, pages 106 – 117, 2005.

[11] Muffy Calder and Stephen Gilmore, editors. *Computational Methods in Systems Biology, International Conference, CMSB 2007, Edinburgh, Scotland, September 20-21, 2007, Proceedings*, volume 4695 of *Lecture Notes in Computer Science*. Springer, 2007.

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

[13] Luca Cardelli. Brane calculi: Interactions of biological membranes. In Vincent Danos and Vincent Schachter, editors, *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2004.

[14] Luca Cardelli. Strand algebras for dna computing. In Deaton and Suyama [24], pages 12–24.

[15] Luca Cardelli and Gheorghe Păun. An universality result for a (mem)brane calculus based on mate/drip operations. In *Proceedings of the ESF Exploratory Workshop on Cellular Computing (Complexity Aspects), Sevilla (Spain), January 31st - February 2nd*, pages 75–94, 2005.

[16] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[17] Alberto Credi, Marco Garavelli, Cosimo Laneve, Sylvain Pradalier, Serena Silvi, and Gianluigi Zavattaro. Modelization and simulation of nano devices in $\{nano\}\kappa$ calculus. In Calder and Gilmore [11], pages 168–183.

[18] Pierre-Louis Curien, Vincent Danos, Jean Krivine, and Min Zhang. Computational self-assembly. *Theor. Comput. Sci.*, 404(1-2):61–75, 2008.

[19] Vincent Danos. Agile modelling of cellular signalling (invited paper). *Electron. Notes Theor. Comput. Sci.*, 229(4):3–10, 2009.

[20] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Scalable simulation of cellular signaling networks. In *APLAS*, pages 139–157, 2007.

[21] Vincent Danos and Cosimo Laneve. Graphs for core molecular biology. In *CMSB*, pages 34–46, 2003.

[22] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

[23] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[24] Russell J. Deaton and Akira Suyama, editors. *DNA Computing and Molecular Programming, 15th International Conference, DNA 15, Fayetteville, AR, USA, June 8-11, 2009, Revised Selected Papers*, volume 5877 of *Lecture Notes in Computer Science*. Springer, 2009.

[25] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[26] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL*, pages 372–385, 1996.

[27] Nissim Francez and Michael Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *FOCS* [2], pages 373–379.

[28] S. Ginsburg. *The Mathematical Theory of Context.Free Languages.* McGraw-Hill, New York, 1966.

[29] Oscar H. Ibarra, Sara Woodworth, H.-C. Yen, and Zhe Dang. On symport/antiport systems and semilinear sets. In *Proceedings of the 6th International Workshop on Membrane Computing (WMC6)*, pages 312–335, 2005. Lecture Notes in Computer Science, Springer. 2005 (to appear).

[30] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, March 2002.

[31] Kurt W. Kohn and David Botstein. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Mol. Biol. Cell*, 10:2703–2734, 1999.

[32] Cosimo Laneve and Luca Padovani. Smooth orchestrators. In Aceto and Ingólfsdóttir [4], pages 32–46.

[33] Cosimo Laneve and Fabien Tarissan. A simple calculus for proteins and cells. *Theor. Comput. Sci.*, 404(1-2):127–141, 2008.

[34] Cosimo Laneve and Antonio Vitale. Expressivity in the kappa family. *Electr. Notes Theor. Comput. Sci.*, 218:97–109, 2008.

[35] Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[36] Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

[37] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[38] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[39] Robin Milner. *Communication and Concurrency*. International Series on Computer Science. Prentice Hall, 1989.

[40] Robin Milner. The polyadic π-calculus: a tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993. An earlier version of this paper appeared as Technical Report ECS-LFCS-91-180 of University of Edinburgh, 1991.

[41] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[42] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Inf. Comput.*, 100(1):1–77, 1992.

[43] Uwe Nestmann. On the expressive power of joint input. *Electr. Notes Theor. Comput. Sci.*, 16(2), 1998.

[44] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000.

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

[46] Andrei Păun and Gheorghe Păun. The power of communication: P Systems with symport/antiport. *New Generation Computing*, 20(3):295–305, May 2002.

[47] Gheorghe Păun. Computing with membranes. Technical Report 208, Turku Center for Computer Science-TUCS, 1998. (www.tucs.fi).

[48] Gheorghe Păun. Computing with membranes. An introduction. *Bulletin of the EATCS*, (67):139–152, February 1999.

[49] Gheorghe Păun. *Membrane Computing. An Introduction*. Springer-Verlag, Berlin, 2002.

[50] Gheorghe Păun. Membrane computing. In Andrzej Lingas and Bengt J. Nilsson, editors, *Fundamentals of Computation Theory 14th International Symposium, FCT*

2003, Malmö, Sweden, August 12-15, 2003, Proceedings., volume 2751 of *Lecture Notes in Computer Science*, pages 284–295. Springer, 2003.

[51] Gheorghe Păun. Introduction to membrane computing. In *First brainstorming Workshop on Uncertainty in Membrane Computing, Palma de Mallorca, Spain, November 2004*, 2004.

[52] Gheorghe Păun. One more universality result for P systems with objects on membranes. In *Proceedings of the Third Brainstorming Week on Membrane Computing, Sevilla (Spain), January 31st - February 4th*, pages 263–274, 2005.

[53] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[54] Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

[55] Corrado Priami. Stochastic pi-calculus with general distributions. In *in Proc. of the 4th Workshop on Process Algebras and Performance Modelling (PAPM '96), CLUT*, pages 41–57, 1996.

[56] A. Regev and E. Shapiro. Cellular abstractions: Cells as computation. *Nature*, 419(6905):343, 2002.

[57] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. *Pac Symp Biocomput*, pages 459–470, 2001.

[58] Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Language*, volume I-II-III. Springer Verlag, Berlin Heidelberg, 1997.

[59] Maria Grazia Vigliotti, Iain Phillips, and Catuscia Palamidessi. Separation results via leader election problems. In de Boer et al. [23], pages 172–194.

[60] Maria Grazia Vigliotti, Iain Phillips, and Catuscia Palamidessi. Tutorial on separation results in process calculi via leader election problems. *Theor. Comput. Sci.*, 388(1-3):267–289, 2007.

[61] A Vitale, G Mauri, and C Zandron. Simulation of a bounded symport/antiport p systems with brane calculi. *Biosystems*, 91(3):558 – 571, 2008.

[62] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.