Università degli Studi di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Informatica
Anno Accademico 2002-2003

Tesi di Laurea Specialistica

A Temporal Logic for HD-Automata

Vincenzo Ciancia

Relatore:
Prof. Ugo Montanari

Controrelatore:
Prof. Andrea Maggiolo Schettini

Contents

Chapter 1

Introduction

Due to the broad diffusion of every kind of network and mobile device in the last few years, computing is rapidly evolving towards what is now called *global computing*. With this term we refer to a new field of research and development in computer science, with innovative features with respects to standard development processes and software architectures:

- computing is *distributed*: rather than a set of libraries, the software designer can use and integrate *services* that frequently are not physically close to each other;

- computing is *mobile*: besides being executed on single machines, with the typical services of a computer, software components also run on a variety of devices, ranging from mobile phones to car equipment, and communication channels change rapidly as soon as the devices move from one location to another;

- programs are *heterogeneous*: components that constitute a software system cannot be assumed to be implemented all in the same language, or using the same technology; *middleware* that allows uniform interaction is introduced, and even this middleware can often be layered into different integration technologies;

- systems are *open-ended*: the designer is no longer required, nor able, to predict and classify all the features that a system can provide or use, since the whole system is not programmed by a single entity; new software components can be added to the system at any time, and already running programs should become aware of their existence, and become able to make use of their functionalities;

– computational entities are *autonomous*: as programs are distributed, and components are mobile, there can't be a global entity controlling their behavior, and there are many independently controlled *software agents* that move from one location to another, enter or leave security domains, access simultaneously many different services, leading to security concerns such as information leaks or access control to private resources, and also to a considerably higher difficulty in designing communication protocols.

To completely meet these goals in software production, work is required from the theoretical point of view, both to develop new models of computation that satisfy the given requirements, and to learn how to reason about the behavior, and check properties, of such systems. In this thesis, we will concentrate on a particular operational model for concurrent, mobile processes, develop a decidable logic to reason about properties of software components represented in this model, and describe how it is possible to algorithmically check if these properties hold.

Formally representing mobility

The research on the formal semantics of concurrent programs had a turn, at the end of the eighties, with the definition of the π-calculus [16]. The π-calculus is a language expressly designed for mobility, with constructs for parallel composition, nondeterministic choice and fresh name generation among others. All previous models for concurrency, such as *Petri nets* [22] or *CCS* [12], lacked the fundamental ability of dynamically reconfiguring the communication structure, which is an essential feature when representing mobility, because it allows processes to switch their communication channel while communication is in progress, without having to interrupt (and establish again) their connection.

Models for concurrent mobile processes are undoubtedly a need in today's computer science, due to the increasing number of systems and protocols running on mobile hardware, but also to the massive usage of code migration mechanisms in distributed and network programming. Actually, π-calculus dynamic reconfiguration of channels is semantically equivalent to having higher-order concurrent processes, as shown by Sangiorgi [23]; this is perhaps a bit surprising, but reflects a view of distributed programming in which we do not know or care about *where* the code is located, but only *how* to reach another process, i.e. the channel by which we can send and receive messages to another part of the program.

The novel approach of the π-calculus was to allow both the transmission of channels as messages, which allows reconfiguration, and the restriction operator (already found in CCS) which corresponds to the generation of a fresh, private channel that can then be communicated to other processes; the calculus was based on *pure* names, in the sense that a name does not designate additional data, but it is simply an abstract notion of an identifier equipped with an equality operation.

According to what Milner itself says [15], names are the most important feature of the π-calculus: apart from the fact that arbitrary data structures can be encoded as π-calculus processes, it is name passing that allows for code mobility and dynamic modification of the connection structure; moreover, the conjunction of this ability with fresh name generation (or restriction) is responsible for much of the expressive power of this language: for example, it allows for atomic communication of sequences of messages, encoding function calls and thus the λ-calculus [13], and is able to represent the unique identity of objects in object-oriented programming, as found in [24], where a complete encoding of a parallel object-oriented language into the π-calculus is introduced. Another key use for the restriction operator is expressing secrecy properties in secure communication protocols: the spi-calculus, developed by Abadi and Gordon in 1996 [1], is an extension of the π-calculus designed for the description and analysis of cryptographic protocols.

After the π-calculus was defined, a great number of variants, extensions and related languages have been invented, and there is an entire family of *calculi* for mobile processes, each one giving different visibility in the operational semantics to particular characteristics such as *locations* or resource usage.

Model checking as feasible mechanical validation of design against requirements

Once found the tools to mathematically represent mobile processes, a very good reason to give a formal semantics to any kind of language or system is to be able to *formally reason* about programs behavior. To this aim, logics are defined, which can express desired properties about systems; these logics are often *modal*, meaning that there are formulae which have a *context*, defining e.g. what has to happen before the given formula can become true. Compare for example what different meaning the assertions "Mario is at home" and "after traveling, Mario is at home" have: the context "after traveling" radically changes the extent of the proposition; in modal logics such a context is called "modality". There are many different kinds of modal logics, depending on what the modalities describe: we range from *temporal* logics, which describe properties of systems as they evolve in time, or *spatial* logics which reason about the structure of system states, such as the number of involved processes or the usage of resources at a specific moment, to more sophisticated forms of reasoning where modalities may indicate *beliefs* of agents regarding to what other agents *know*, or actions which are executed with given probabilities.

Having defined any kind of logic for a language, one needs an interpretation relation between formulae and states, or programs, which explains *how* to check if a formula holds. It is desirable, at this point, to introduce some mechanized procedure to check the truth of these assertions. *Theorem provers* and *model*

checkers are semi-automatic or fully-automatic tools to accomplish this goal.

In practice, a model checker works by enumeration of all the possible states of a system, thus it is implicitly decidable if the state space is finite and the interesting properties are decidable, as it is often the case, and the verification procedure can be fully mechanized. On the other hand, theorem proving is a much more powerful and general technique, which can check properties of infinite systems, but requires that some parts of the proof be done "by hand" because the properties of interest are often not decidable or even not semidecidable. In fact, there are no fully automatic theorem provers, but only *proof assistants* or verifiers, that check if a proof, in part constructed by hand, is correct. So theorem proving it's not usable by any programmer or software engineer, but requires experience with mathematical logics, and theorem proving skills, to be used. Model checking appears to be the right compromise between verification possibilities and real world necessities, as underlined in [2]. A more complete comparison of these two approaches to formal verification can be found in [9].

Generally speaking, what many formal reasoning methods try to achieve is to define a *design* language, not an implementation one, for certain parts of a system, and then to validate part of the project requirements by expressing them into a modal logic. This has many advantages over a full verification of an entire software product:

- the approach is practically feasible, since design languages usually describe few, interesting properties of a system (e.g. its behavior w.r.t. concurrency and communication), while often full verification is impossible due to the size of the implementation
- we may validate the design against requirements before the implementation effort takes place
- model checking algorithms usually return a counterexample when a formula is not verified by the system, so we can discover and examine in detail, in the form of use cases, many potential problems with the design, which might be difficult to address in other stages of the production cycle
- if a property is not satisfied, this may as well indicate that the system is designed the right way, but the requirement is too strong, i.e. formal reasoning can help *validating* requirements and checking that requirements are really what we expect from the system.

Model checking is decidable for large classes of programs, thus it is relatively easy to build automatic model checkers. Complexity is polynomial in terms of the size of the state space of the system and of the formula being checked, and

so can be exponential in the size of the programs, but there are a lot of practical cases in which this method proves efficient, because the state space is relatively small (again, because the design simplifies over many properties of a system, which are not interesting from a global point of view). This technique is already used in a great deal of applications, ranging from business protocols to hardware verification, and there are examples of security protocols vulnerable to attacks that were only discovered, after years from the definition of the protocol itself, by the use of a model checker (the typical example is the *Needham-Schroeder* protocol [10]).

It may seem right now that model checking is the magic wand by which we can verify any requirement expressible into modal logics. In fact, even if this procedure is often decidable, there is an important concern, which obviously is in the size of the state space being checked. Realistic applications of model checking usually include the so called "control" problems, where it does not matter what data comes through the system, but how this data is created, routed and collected. Examples of this kind of problems are, again, communication protocols, but also hardware verification, and less computer-science related topics such as workflow management, where formal methods, used to model passing of tasks and control between workgroups, are assuming a crucial role, due to the growing size of business companies and their departments.

HD-automata: towards an universal operational model

Operational semantics, in the case of mobile processes, frequently gives rise to non-finitary transition systems: for example, in the standard LTS (labeled transition systems) semantics for the π-calculus, the input action is infinite-branching, i.e. there are infinitely many possible events when an agent tries to read a value from a channel, one for each possible name that could be received, thus making exhaustive model checking undecidable even for very simple programs.

Better operational models, which are finite-branching for interesting classes of π-calculus programs, have been proposed. History-dependent automata (HD-automata for short) are a particular operational model similar to labeled transition systems, where states are enriched with additional information that depends on previous transitions that the system has made (hence the attribute *history-dependent* in their name). In this thesis, we will use *history-dependent automata with symmetries*, where this additional information is encoded in the form of symmetries (sets of permutations that are groups w.r.t. composition) over names.

History-dependent automata, as we will see, have many useful characteristics that make them a good model for concurrency and mobility. First of all, finite-state models are obtained from useful categories of π-calculus agents, thus allowing

model checking; besides, not only π-calculus, but also other formalisms for concurrency, including Petri nets [20] and calculi with localities [18], can be modeled adequately. In general, it seems that calculi with name passing (or binding) and name generation can be translated into HD-automata.

Working with an unified model can lead to a significant simplification in the development of tools for formal design and verification of concurrent systems. If written for HD-automata, those need not to be adapted to many different process calculi. Another important property of HD-automata with symmetries is that they can be minimized [7], allowing optimal model and bisimilarity (a notion of behavioral equivalence strictly linked to the operational semantics) checking. The ability of being minimized is not a feature of every model for mobile processes, and in particular minimization of π-calculus agents required much work on the theoretical side. In [19] a coalgebraic operational semantics with minimization for the π-calculus is given, and it's strictly linked with the class of HD-automata with symmetries used in this work.

Previous work on the subject and aim of the thesis

The logic for the π-calculus described here is directly derived from the seminal work of Milner, Parrow and Walker [17], where it is shown what relations hold between variants of the π-calculus semantics and different modal logics for mobile processes. In [17] no fixpoint operator, or path operator, is given, while we will extend our work to a simple but powerful path modality. Perhaps the most powerful temporal logic is the propositional μ-calculus [3], a modal logic with name passing and two fixpoint operators (*least* and *greatest* fixpoint). Regarding HD-automata, there have been previous works, both for other process calculi, as in [25], where a simpler history-dependent operational model for Petri nets is given together with an adequate modal logic, and for the π-calculus itself, as in the π-logic [8, 5], that already uses HD-automata to verify π-calculus agents properties.

The developments of our work, including the proof of adequacy, are new w.r.t. [17], because of the presence of symmetries into HD-automata, and also because of the different fresh name generation mechanism which the HD model has, if compared to the π-calculus. Even previous results found in [8] could not have been reused here, because adequacy results are proved for the π-logic, and not for any HD-automata logic; moreover, the flavor of HD-automata used in [8] is the one without symmetries; even if in the "if" part of the proof of adequacy we use the same concept of [17], symmetries in states change the way proofs are done and so we had to work out these from scratch.

There are many good-working implementations of model checkers. One example, suitable to check properties of π-calculus and fusion calculus programs, is the

"Mobility Workbench" [27], a powerful tool to verify even the open semantics of the π-calculus. Another important operational environment for mobile processes is the "HD-automata Laboratory" [26], which implements the framework for the π-logic cited above, that we are going to extend with results from this work.

Here we introduce a modal logic for mobile processes, and give an interpretation both over the π-calculus and over *history-dependent automata with symmetries*. The main point is that the interpretation over HD-automata is generalizable to any modal logic with name generation and name passing, so that it is possible to reuse algorithms developed here to model check, for example, Petri nets, by first translating a net into a HD-automata and then checking logic formulae over this model.

The novelty of our approach, again, is in the quest for an unique operational model, which can be minimized and checked, reusable for various kinds of systems. An evidence of HD-automata with symmetries being the right choice for this purpose, apart from the many process calculi that already can be translated into this formalism, is that in the definition of the logic we have no side conditions for modalities: every necessary condition over names is already encoded in the mere presence of a transition with the desired label, i.e. HD-automata transitions carry enough information to check temporal logics with name passing and generation. Thus the theory is much simplified w.r.t. the one introduced for the π-calculus, and this is promising from an implementation point of view, since no special cases have to be made (e.g. for name generation) in the model checking algorithm.

The thesis is divided into six chapters. Chapter 2 introduces the π-calculus and HD-automata with symmetries, chapter 3 is the core of the work, describing in detail a modal logic for the π-calculus, an interpretation over HD-automata, explaining how we went to such a definition, and providing proofs of adequacy, soundness and completeness.

The logic is first introduced as a pure Hennessy-Milner one, then an *eventually* modality is added; this has been made both to add expressive power, since the enriched logic can express *liveness* (something good *eventually* happens) and *safety* (something bad *never* happens) properties, and to show that it is feasible to add fixpoint-alike operators.

The π-calculus semantics mainly used in this work is the *early* semantics, but in chapter 3 we also give an account of how to treat the *late* semantics, and explain how the results of this thesis already hold for the late variant, too, by exploiting the translation of the late semantics into the early one found in [6]; this translation is done adding two new kinds of transitions to the early operational semantics, the second always used immediately after the first, which correspond to preparing to receive input on a certain channel, and then reading a value from the same channel. In [17] it is shown that to characterize late bisimulation, a logic needs to have both a *bound input modality* and a name matching construct.

It is clear that these two result are very similar: the two added transitions of [6], if translated into modalities, are a form of bound input immediately followed by a specialized matching construct. This is explained in more detail in the section about the late semantics.

In chapter 4 we describe the model checking algorithm, explaining why it is decidable (if the state space is finite) and describing an efficient model checking procedure. Case studies showing the expressive power of the logic are given in chapter 5.

Chapter 2

Preliminaries

2.1 The π-calculus

The π-calculus has been presented by Milner, Parrow and Walker in 1992. It is a process algebra featuring concurrency and nondeterminism, and being a nominal calculus it has name generation, name passing and a notion of equality over names. As we already said, the novelty of this language was the introduction of name passing, which allows for reconfiguration of the communication structure at runtime. Since its introduction it has been object of intensive study, both to test its expressive power and limitations, and to exploit practical possibilities of formal methods applied to mobility. In fact, the large diffusion of systems for distributed programming, often in the form of *client/server* programs for the *world wide web*, has given rise to much interest in this field of research. It has been discovered that the combination of name generation and passing can express a great deal of programming language features, ranging from unique objects to function and method calls.

In an operational model (e.g. labeled transition systems) the operational semantics naturally gives rise to a notion of *bisimilarity* or *behavioral equivalence*, which asserts that two programs perform the same transitions, and the destination states of these transitions are still equivalent.

There are many operational semantics, and thus bisimulation equivalences, for the π-calculus: the semantics can be *synchronous* or *asynchronous*, *early*, *late* or *open*, and there are even more variants taking in account localities or spatial properties of states (such as resource usage). Our version will be the early, synchronous π-calculus, where the term *early* refers to the instantiation time of the received name in an input action (see [17] for a detailed explanation of the differences between early and late bisimulation).

2.1.1 Syntax

Given a denumerable infinite set of *names* $\mathfrak{N} = \{x_0, x_1, x_2, \ldots\}$, the set of π-*calculus agents* is defined by the syntax

$$P ::= \mathbf{0} \;\Big|\; \alpha.P \;\Big|\; P_1 \mid P_2 \;\Big|\; P_1 + P_2 \;\Big|\; (\boldsymbol{\nu} x)\ P \;\Big|\; [x = y]\, P \;\Big|\; A\left(x_1, \ldots, x_{r(A)}\right)$$

$$\alpha ::= \tau \;\Big|\; x(y) \;\Big|\; \overline{x}y$$

where $r(A)$ is the rank of the *process identifier* A. Free names and bound names of an agent P are defined as usual and indicated with $fn\,(P)$ and $bn\,(P)$, with $n(P) = fn\,(P) \cup bn\,(P)$; name y is bound in $x(y).P$ and $(\boldsymbol{\nu} y)\ P$.

Each identifier A has a definition $A\left(y_1, \ldots, y_{r(A)}\right) = P_A$ (with y_i all distinct and $fn\,(P_A) \subseteq \left\{y_1 \ldots y_{r(A)}\right\}$) and each identifier in P_A is in the scope of a prefix (guarded recursion).

We define an equivalence relation over π-calculus agents, called *structural congruence*, which is the smallest congruence that satisfies the following axioms, and identify agents up to this equivalence:

(alpha) $P \equiv Q$ if P and Q are alpha equivalent

(sum) $P{+}\mathbf{0} \equiv P \qquad P{+}Q \equiv Q{+}P \qquad P{+}(Q{+}R) \equiv (P{+}Q){+}R$

(par) $P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R$

(res) $(\boldsymbol{\nu} x)\ \mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu} x)\ (\boldsymbol{\nu} y)\ P \equiv (\boldsymbol{\nu} y)\ (\boldsymbol{\nu} x)\ P$

$(\boldsymbol{\nu} x)\ (P|Q) \equiv P|\,(\boldsymbol{\nu} x)\ Q \quad \text{if } x \notin fn\,(P)$

(match) $[x{=}y]P \equiv [y{=}x]P$

A quick review of the syntax will serve us to informally explain the meaning of the constructs:

- the inactive process is called $\mathbf{0}$ and does nothing;
- the syntactical category α is that of the so-called *prefixes*; τ is the prefix indicating the unobservable action, corresponding to an internal computation, $x(y)$ is the input prefix, corresponding to waiting for a name to be

sent over channel x, and then reading it into the local variable y for further usage; finally, $\overline{x}y$ is the output prefix, corresponding to sending the name y over the channel x (there is no distinction between a name and a channel, and both the terms are to be considered synonyms when working with π-calculus);

- the process $\alpha.P$ performs the action corresponding to α and then *behaves* like P;
- the construct + is nondeterministic choice: $P + Q$ can behave as either P or Q;
- the construct | is parallel composition: $P|Q$ intuitively "runs" P and Q in parallel, allowing them to synchronize by message passing over channels;
- $(\boldsymbol{\nu}x)\ P$ is the *restriction* operator: it makes the name x a local name which other processes do not know; it is easier to read $(\boldsymbol{\nu}x)\ P$ as "create a fresh channel named x and then act like P";
- $[x{=}y]P$ is the *matching* construct: the obtained process can behave like P, but only when x and y are equal;
- finally, an iterative behavior can be expressed with the recursive definition construct. The original version of the π-calculus had a different mechanism for this purpose, the *bang* or *replication* operator, which roughly corresponds to spawning infinite copies of a process; the two methods have the same expressive power but using named recursion has the advantage of being able to define syntactical classes of agents which have a finite-state behavior (e.g. the class of *finite control* agents defined below).

2.1.2 Early operational semantics

The operational semantics is given in terms of what actions a process performs; actions are different from prefixes, even if they look similar (indeed, prefixes are a syntactical domain, while actions are the corresponding semantical domain). An action can be, in the early operational semantics, one of:

τ: this stands for an internal computation which the agent performs;

xz: this represents free input, which is the reception of the name z over the channel x. The difference between the free input action and the bound input prefix is that here the name z is *free*, i.e. it is not a placeholder but the real name received. In fact, this means that there are infinitely many branches in the LTS for the operational semantics of an input action;

$\overline{x}z$**:** this represents free output, i.e. the process is sending the name z over the channel x;

$\overline{x}(z)$**:** this is the bound output actions, that happens when a process transmits over channel x a name that it has just generated, as in $(\nu y)\overline{x}y.\mathbf{0}$, which generates y and then communicates it over channel x.

We can now give the operational semantics of an agent in the form of a labeled transition system, in small groups of rules which we comment separately. Besides the rules of the operational semantics, there is an implicit *structural congruence axiom*, which states that structurally equivalent π-calculus agents have the same transitions. Since we silently identify structurally congruent agents we do not have to add this axiom to the operational semantics, but it is useful to keep it in mind:

$$\frac{P' \equiv P, P \xrightarrow{\alpha} Q, Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

– $\tau.P \xrightarrow{\tau} P \qquad \bar{x}y.P \xrightarrow{\bar{x}y} P \qquad x(y).P \xrightarrow{xz} P\{z/y\}$

These are the simplest rules. Note that for bound input the name z can be any name, either free in the agent or unknown to it.

– $\dfrac{P_1 \xrightarrow{\alpha} P'}{P_1+P_2 \xrightarrow{\alpha} P'} \qquad \dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2}$ if $bn(\alpha) \cap fn(P_2) = \emptyset$

The rule for nondeterministic choice (and other symmetric rules) are given only for one of the two possible choices, because the other one is given by the structural congruence axiom. In the first rule for parallel composition, we have a side condition. Two parallel processes can proceed independently, but if one of the two generates a fresh name, this name has to be fresh also for the other one, hence the side condition, which is however not limiting the possible transitions of the first process, because it is α-convertible to avoid this "fresh name capture".

– $\dfrac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'} \qquad \dfrac{P_1 \xrightarrow{\bar{x}(y)} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} (\boldsymbol{\nu} y)\ (P_1'|P_2')}$ if $y \notin fn(P_2)$

The left rule is called *synchronization*: process P_1 is offering a synchronous communication to the outside world, and P_2 is attempting to receive a value from the same channel. The two process synchronize, and the observable result is a τ transition. The right rule represents transmission of a fresh name to another process: P_2 gains the new name, but it must obey to the same rules as if it had generated the name, thus it's put inside the scope

of the restriction. When iterated, this method ensures that a process P_j receiving a name which is fresh in P_i, either because P_i has generated it or because P_i has received it from another process, will remain fresh in all the processes. This rule, together with the first rule for parallel, is responsible for the power of the π-calculus, but also for the difficulties when handling its operational semantics (in a global environment, it's not trivial to ensure that a given name is not known to any other process).

– $\dfrac{P \xrightarrow{\bar{x}y} P'}{(\boldsymbol{\nu} y)\ P \xrightarrow{\bar{x}(y)} P'}$ if $x \neq y$ $\qquad$ $\dfrac{P \xrightarrow{\alpha} P'}{(\boldsymbol{\nu} x)\ P \xrightarrow{\alpha} (\boldsymbol{\nu} x)\ P'}$ if $x \notin n(\alpha)$

Here we see how fresh names are generated and offered for communication, and how a process which has created a fresh name is not compelled to immediately use it: the process can perform any transition not involving the new name.

– $\dfrac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$ $\qquad$ $\dfrac{P_A\{y_1/x_1 \cdots y_n/x_n\} \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_n) \xrightarrow{\alpha} P'}$ if $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$

These are the last two rules, the one for matching and the one for recursion.

We can now introduce the definition of early bisimulation for the π-calculus:

Definition 2.1 (early bisimulation). A relation $\mathcal{R}$ over agents is an *early simulation* if whenever $P \mathcal{R} Q$ then:

– for each $P \xrightarrow{\alpha} P'$ with $bn(\alpha) \cap fn(P, Q) = \emptyset$ there is some $Q \xrightarrow{\alpha} Q'$ such that $P' \mathcal{R} Q'$.

The side condition ensures that there is no free name capture in both processes, but does not affect completeness of the definition, since the label is α-convertible and free names of both the agents are in a finite number.

A relation $\mathcal{R}$ is an *early bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are early simulations.

Two agents P and Q are *early bisimilar*, written $P \sim_\pi Q$ (or simply $P \sim Q$ if it is clear from the context that we are talking about π-calculus bisimilarity) if $P \mathcal{R} Q$ for some early bisimulation $\mathcal{R}$. We call relation $\sim_\pi$ *early bisimilarity*.

2.1.3 Example: buffers and memory cells

A simple synchronous one-position buffer spends its whole life alternating an input state, where it reads an input data, to an output state, where it sends data to another object. This can be modeled in the π-calculus using a single channel ch to communicate with the outside world, as follows:

$$Buf(c) \stackrel{\text{def}}{=} c(x).\overline{c}x.Buf(c)$$

This behavior is different from that of, e.g., a memory cell, which can be either read or written at a given time, in a non deterministic way:

$$\begin{aligned} Cell(c) &\stackrel{\text{def}}{=} c(x).FullCell(c,x) \\ FullCell(c,x) &\stackrel{\text{def}}{=} c(y).FullCell(c,y) + \overline{c}x.FullCell(c,x) \end{aligned}$$

2.2 Names and Permutations

A *name substitution* is a function $\sigma : \mathfrak{N} \to \mathfrak{N}$. We will write $\sigma;\sigma'$ for the reverse composition of substitutions σ and σ'; that is, $(\sigma;\sigma')(x) = \sigma'(\sigma(x))$. We denote with $\{y_1/x_1 \cdots y_n/x_n\}$ the substitution that maps x_i into y_i for $i = 1, \ldots, n$ and which is the identity on the other names.

A *name permutation* is a bijective name substitution; we first need to define an operator over a permutation ρ which shifts it up n names; we will use the notation ρ_{+n}:

$$\rho_{+n}(x_i) = \begin{cases} x_i & \text{if } i < n \\ x_{j+n} & \text{if } \rho(x_{i-n}) = x_j \end{cases}$$

It's easy to show that $(\sigma;\rho)_{+n} = (\sigma_{+n};\rho_{+n})$ and that $(\sigma^{-1})_{+n} = (\sigma_{+n})^{-1}$; moreover, $\sigma_{+0} = \sigma$.

The *kernel* $K(\rho)$ of a permutation ρ is the set of the names that are changed by the permutation:

$$K(\rho) = \{n \,|\, \rho(n) \neq n\}.$$

Permutation ρ has *finite kernel* if $K(\rho)$ is finite. A finite-kernel permutation can change only a finite subset of the names. In particular, the identity substitution id is a finite-kernel permutation with $K(id) = \emptyset$.

A *symmetry* S is a group of finite-kernel permutations, i.e. a set of finite-kernel permutations closed for composition. Since permutations are finite-kernel, the identity and inverse permutations are necessarily present in a symmetry under this condition. We denote with $\mathcal{S}ym$ the set of the symmetries on $\mathfrak{N}$.

A symmetry S has *finite support* if it contains all the permutations that do not change the names of a certain finite set (but it does not necessarily contain *all* the permutations that change it):

$$\exists N \subset \mathfrak{N}.\forall \rho.(\forall n \in N.\rho(n) = n) \Rightarrow \rho \in S$$

In this case, the *support* of S, written $supp(S)$, is the smallest set of names that satisfies this condition.

If a symmetry S has finite support, then it can be described as follows:

$$S = \{\rho'; \rho \,|\, \rho \in S \wedge (\forall n \notin supp(S).\ \rho(n) = n) \wedge (\forall n \in supp(S).\ \rho'(n) = n)\}.$$

That is, the permutations of symmetry S can be obtained, from the permutations $\rho \in S$ that act only on the names in the support, by applying the permutations on the names outside the support. This implies that we can finitely represent a finite support symmetry S by giving its support $supp(S)$ and the set $ssym(S)$ of permutations that only act on $supp(S)$.

2.3 History-dependent automata with symmetries

History-dependent automata with symmetries [18, 19] allow the representation of the behavior of concurrent systems up to name permutations, by enriching with finite support symmetries the information contained in each state of the transition system corresponding to the system semantics. Names appearing in states do not have a global meaning, but only a local identity, which maps to the previous state, revealing where a name came from, by the means of a permutation associated to each transition. In HD-automata, transitions implicitly carry additional information; for example, we never encode non-representative π-calculus transitions in HD transitions, but only the representative ones, since we can encode all the non representative transitions by using symmetries associated to the states.

We will not present here the full theory of HD-automata, but we just say that for any HD-automata there exists a minimal automata bisimilar to it, and that there exist finite algorithms to perform minimization.

Definition 2.2 (HD-automata). A *HD-automaton with Symmetries* (or simply *HD-automaton*) $\mathcal{A}$ is a tuple $\langle \mathcal{S}, sym, \mathfrak{L}, \longmapsto \rangle$, where:

- $\mathcal{S}$ is the set of *states*;
- $sym : \mathcal{S} \to \mathcal{S}ym$ associates to each state a finite support *symmetry*;
- $\mathfrak{L}$ is the set of *labels*;

– $\longmapsto \subseteq \{\langle Q, l, \zeta, Q'\rangle \,|\, Q, Q' \in \mathcal{S},\ l \in \mathfrak{L},\ \zeta \text{ is a finite-kernel permutation}\}$ is the *transition relation*, where:

  – Q and Q' are, respectively, the source and the target states;
  – l is the label of the transition, and
  – ζ is a permutation mapping names of the target state into names of the source state of the transition.

  Whenever $\langle Q, l, \zeta, Q'\rangle \in \longmapsto$ then we write $Q \overset{l}{\underset{\zeta}{\mapsto}} Q'$.

We assume that each label $l \in \mathfrak{L}$ has the form $\alpha(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n \in \mathfrak{N}$, and that each label is in a set $\mathfrak{L}_i$, where $i \in \mathbb{N}$ represents the number of *fresh* names generated along a transition. Note that fresh name generation is an intrinsic feature for the HD-automata model, and this greatly decreases the complexity of mappings for calculi with names into HD-automata. Fresh names are not mentioned in the arguments of a label, but rather free names of the source state of a transition are shifted (in the sense of sending x_0 into x_1, x_1 into x_2 and so on) in the destination state to make place for the new ones, the number of indexes to shift being given by the index of the set $\mathfrak{L}_i$ of the label. This machinery, analogous to *de Bruijn indices* [4], will be clearer when we will see how π-calculus agents are translated into HD-automata.

We do not allow distinct isomorphic transitions between the same states to be present in a HD-automaton, where $Q \overset{l_1}{\underset{\zeta_1}{\mapsto}} Q'$ and $Q \overset{l_2}{\underset{\zeta_2}{\mapsto}} Q'$ are isomorphic if there exists some $\rho \in sym(Q)$ such that:

– $l_2 = \rho(l_1)$;

– $\zeta_1; \rho_{+n}; \zeta_2^{-1} \in sym(Q')$ if $l_1 \in \mathfrak{L}_n$.

We now define HD-bisimulation. Since the names that appear in the states of the HD-automata have only a local identity, a HD-bisimulation is defined as a set of triples $\langle Q_1, \delta, Q_2\rangle$, where δ is a permutation of $\mathfrak{N}$ that sets a correspondence between the names of Q_1 and those of Q_2.

Definition 2.3 (HD-bisimulation). Let $\mathcal{A}$ be a HD-automaton. A *HD-simulation* for $\mathcal{A}$ is a set of triples

$$\mathcal{R} \subseteq \{\langle Q_1, \delta, Q_2\rangle \,|\, Q_1, Q_2 \in \mathcal{S},\ \delta \text{ is a finite-kernel permutation}\}$$

such that, whenever $\langle Q_1, \delta, Q_2\rangle \in \mathcal{R}$ then:

– for each $\rho_1 \in sym(Q_1)$ and each $Q_1 \overset{l_1}{\underset{\zeta_1}{\longmapsto}} Q_1'$, there exist some $\rho_2 \in sym(Q_2)$ and some $Q_2 \overset{l_2}{\underset{\zeta_2}{\longmapsto}} Q_2'$, such that:

  – $l_2 = \gamma(l_1)$, where $\gamma = \rho_1; \delta; \rho_2^{-1}$
  – $\langle Q_1', \delta', Q_2' \rangle \in \mathcal{R}$, where $\delta' = \zeta_1; \gamma_{+n}; \zeta_2^{-1}$ if $l_1 \in \mathfrak{L}_n$.

A *HD-bisimulation* for $\mathcal{A}$ is a set of triples $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1} = \{\langle Q_2, \delta^{-1}, Q_1 \rangle \,|\langle Q_1, \delta, Q_2 \rangle \in \mathcal{R}\}$ are HD-simulations for $\mathcal{A}$.

A single HD-automaton transition represents a whole set of transitions that differ for a permutation in the symmetry of the source state, so that a matching transition must be chosen, among those going out of Q_2, for every different permutation in this set (but for the same reason, permutations can be applied also when looking for a transition from Q_2).

Two transitions $Q_1 \overset{l_1}{\underset{\zeta_1}{\longmapsto}} Q_1'$ and $Q_2 \overset{l_2}{\underset{\zeta_2}{\longmapsto}} Q_2'$ match only if they have the same label up to the appropriate permutation γ, and the target states are related in the HD-bisimulation, via a correspondence δ' that is obtained from γ by applying the name correspondences ζ_1 and ζ_2^{-1}. In the case of transitions generating fresh names, permutation γ is shifted before applying substitutions ζ_1 and ζ_2^{-1}, because the first n names are considered fresh in the destination state of the transition.

Definition 2.4 (largest HD-bisimulation). Let $\mathcal{A}$ be a HD-automaton. We denote with $\mathcal{R}_\mathcal{A}$ the largest HD-bisimulation for $\mathcal{A}$:

$$\mathcal{R}_\mathcal{A} = \bigcup_{\mathcal{R} \text{ is a HD-bisimulation for } \mathcal{A}} \mathcal{R}\,.$$

When $\langle Q, \delta, Q' \rangle \in \mathcal{R}_\mathcal{A}$ we also write $Q \overset{\delta}{\sim} Q'$.

2.4 From π-calculus agents to HD-automata

Here we will explain how a π-calculus agent can be translated into a HD-automata, and tell sufficient conditions for this construction to be finite.

We introduce the function l to translate labels from π-calculus:

$$\begin{aligned} l(\tau) &= tau \in \mathfrak{L}_0 \\ l(xy) &= in(x,y) \in \mathfrak{L}_0 \\ l(\overline{x}y) &= out(x,y) \in \mathfrak{L}_0 \\ l(\overline{x}(y)) &= bout(x) \in \mathfrak{L}_1 \end{aligned}$$

An important thing to notice is that the bound output label has only one argument, corresponding to the channel over which a fresh name is sent. The fresh name itself is always x_0, which implicitly differs from any other name because of the way the translation is made.

We define a substitution which is used to introduce fresh names: this substitution is indexed over a finite set s of n names and sends them (sorted) to $x_0, \ldots, x_{n-1}$, while shifting up n positions the remaining names:

$$rot_s(x_i) = \begin{cases} x_{k-1} & \text{if, after sorting, } x_i \text{ is the } k^{th} \text{ element of } s \\ x_{i+n} & \text{if } x_i \notin s \text{ and } n \text{ is the cardinality of } s \end{cases}$$

It does not really matter in what order we take the n fresh names, since they are new, provided that this order is fixed once for all (for example, as we do in the definition, by exploiting an ordering over names).

It's important to underline that the substitution rot_s is not bijective, so we will have to be careful in not trying to find its inverse, but $rot_{bn(x)}$ restricted to $fn(x)$ is, of course, invertible. It holds that $rot_\emptyset = id$.

In the HD-automaton a single state is used to represent a whole set of agents that differ for name permutations; we define a canonical representative for this set, assuming to have a function *norm* that, given a π-calculus agent P, returns a pair $\langle Q, \rho \rangle = norm\,(P)$, where Q is the representative of the class of agents differing from P for a name permutation which leaves the agent into the same orbit, and ρ is a finite-kernel permutation such that $P = Q\rho$.

We also define *representative transitions*, so that a single transition is taken out of many transitions differing only for the choice of a bound name. Representative transitions are also useful to avoid inserting in the obtained HD-automaton transitions that just differ for permutations in the symmetry of their source state (and would be isomorphic in the HD-automaton).

Definition 2.5 (representative transitions). Let $\sqsubseteq$ be the partial order on π-calculus actions defined as follows:

- $\tau \sqsubseteq \tau$,
- $xy \sqsubseteq wz$ if $x < w$, or if $x = w$ and $y \leq z$,
- $\bar{x}y \sqsubseteq \bar{w}z$ if $x < w$, or if $x = w$ and $y \leq z$,
- $\bar{x}(y) \sqsubseteq \bar{w}(z)$ if $x < w$, or if $x = w$ and $y \leq z$.

A π-calculus transition $P \xrightarrow{\mu} Q$ is a *representative transition* if, whenever $P \xrightarrow{\mu'} Q'$ and there exists some permutation ρ such that

- $P \equiv P\rho$ (i.e., ρ is a permutation in the symmetry of P),
- $\mu' = \rho(\mu)$ and $Q' \equiv Q\rho$ (i.e., the two transitions correspond via ρ),

then relation $\mu \sqsubseteq \mu'$ holds.

Notice that if $P \xrightarrow{\bar{x}(y)} Q$ is a representative transition, then $y = \min(\mathfrak{N} \setminus fn\,(P))$. Analogously, if $P \xrightarrow{xy} Q$ is a representative transition and $y \notin fn\,(P)$, then $y = \min(\mathfrak{N} \setminus fn\,(P))$.

Definition 2.6 (from π-calculus agents to HD-automata). The HD-automaton HD_π corresponding to the π-calculus is defined as follows:

- $\mathcal{S} = \{Q \,|\, \langle Q, \sigma\rangle = norm\,(P) \text{ for some } P \text{ and } \sigma\}$;
- $sym(Q) = \{\rho \,|\, Q \equiv Q\rho\}$ for each $Q \in \mathcal{S}$;
- if $Q \in \mathcal{S}$, $Q \xrightarrow{\mu} Q'$ is a representative transition, and $norm\left(Q\, rot_{bn(\mu)}\right) = \langle Q'', \zeta\rangle$, then $Q \xmapsto[\zeta]{l(\mu)} Q''$.

Let P be a π-calculus agent and let $norm\,(P) = \langle Q, \sigma\rangle$. Then we denote with $HD_\pi(P)$ the HD-automaton corresponding to P. It consists of the subsets of the states and of the transitions of HD-automaton HD_π that are reachable from state Q.

Definition 2.6 produces HD-automata that in general are infinite. However, there are classes of π-calculus agents that generate finite HD-automata: this is case of *finitary* π-calculus agents:

Definition 2.7 (finitary agents). The *degree of parallelism* $\deg(P)$ of a π-calculus agent P is defined as follows:

$$\begin{array}{rclrcl} \deg(\mathbf{0}) & = & 0 & \deg(\mu.P) & = & 1 \\ \deg((\boldsymbol{\nu}x)\ P) & = & \deg(P) & \deg(P|Q) & = & \deg(P)+\deg(Q) \\ \deg([x{=}y]P) & = & \deg(P) & \deg(P{+}Q) & = & \max(\deg(P),\deg(Q)) \\ \deg(A(x_1,\ldots,x_n)) & = & 1 & & & \end{array}$$

A π-calculus agent P is *finitary* if $\max\{\deg(P') \mid P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_i} P'\} < \infty$.

In [19] it is shown that if P is a finitary π-calculus agent then $HD_\pi(P)$ is finite, in the sense that it has finite states and transitions (thanks to the use of finite support symmetries, which also are representable with finite memory).

In general, it is not decidable whether an agent is finitary, but there are subclasses of finitary agents which can be characterized syntactically. As an example we can consider *finite control* agents, which have no parallel composition in the body of recursive definitions. In this case, after an initialization phase during which a finite set of processes acting in parallel is created, no new processes can be generated; therefore a finite-control agent is clearly finitary.

The following theorem, proved in [21] for a slightly different version of HD-automata with symmetries, states that two π-calculus agents are bisimilar, according to the standard π-calculus semantics, if and only if the corresponding states in HD-automaton HD_π are HD-bisimilar.

Theorem 2.8. Let P_1 and P_2 be two π-calculus agents and let $norm\,(P_1) = \langle Q_1, \sigma_1\rangle$ and $norm\,(P_2) = \langle Q_2, \sigma_2\rangle$. Then $P_1 \sim_\pi P_2$ iff $Q_1 \overset{\delta}{\sim} Q_2$, where $\delta = \sigma_1; \sigma_2^{-1}$

Clearly, when HD-automata are exploited to check whether two π-calculus agents P_1 and P_2 are bisimilar, it is not necessary to generate the whole HD-automaton HD_π, since is sufficient to generate the subset of the states reachable from Q_1 and Q_2. We already know that the obtained HD-automaton is finite whenever agents P_1 and P_2 are finitary. Therefore, HD-automata can be effectively used to check π-calculus bisimilarity for finitary agents.

2.4.1 Example: translating *buf* and *cell*

Now we see how to translate the one-position buffer and the memory cell of section 2.1.3. Note that our names are sorted as in the English alphabet, so the

first name is a, and the rest are $b, c, \ldots$. Using this convention instead of $x_0, x_1, \ldots$ will ease reading. Also note that the normalization function that is applied to every destination state of a representative π-calculus transition can replace bound names, but the backward substitution which is present in every transition of an HD automata won't, because a substitution is not applied to bound names, thus agents are silently identified modulo α-equivalence.

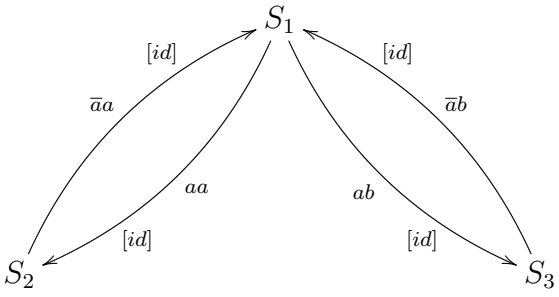
1. Normalize the buffer:
   $norm(Buf(c)) = norm(c(x).\overline{c}x.Buf(c)) = \langle a(b).\overline{a}b.Buf(a), \{c/a, a/c\}\rangle$
2. Find the outgoing π-calculus representative transitions from the initial state:

$$a(b).\overline{a}b.Buf(a) \xrightarrow{aa} \overline{a}a.Buf(a)$$

$$a(b).\overline{a}b.Buf(a) \xrightarrow{ab} \overline{a}b.Buf(a)$$

   These are the only two representative transitions, because $b = min(\mathfrak{N} \setminus fn\,(Buf(a)))$.
3. Iterate the procedure, by normalizing destination states, and annotating the transitions with the mapping obtained from normalization.

We draw the HD-automaton corresponding to our buffer, which has three states, using square brackets to visually distinguish name mappings from actions:



State S_1**:** $a(b)\overline{a}b.Buf(a)$, **support**: {a}, **suppsym**: {}

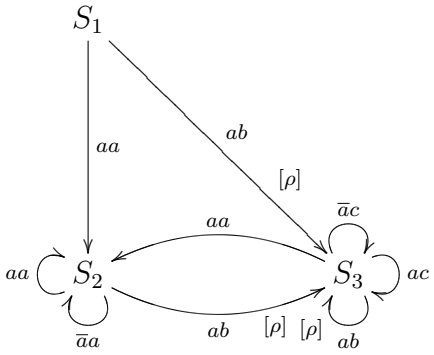
State S_2**:** $\overline{a}a.Buf(a)$, **support**: {a}, **suppsym**: {}

State S_3**:** $\overline{a}b.Buf(a)$, **support**: {a,b}, **suppsym**: {}

We also should have annotated each state with its symmetry; according to equation 2.2 we just have to give the support and the permutations that act over it; in the case of $Buf(a)$ the support is only a and there are, of course, no permutations which change it. State 2 is in the same situation while in state 3 there are again no permutations in the symmetry, but the support contains both a and b.

Notice how the only thing that matters about names, in HD-automata, is their identity: there are two very similar states (state 2 and 3), one where the two free names coincide, and the other one where they are different. There is no other information that we can get about the identity of a name in a state (but transitions record the origin of names in the global namespace).

Here is the HD-automaton corresponding to the memory cell after normalization; the identity mappings have been omitted, while $\rho = \{b/c, c/b\}$:



State S_1**:** $a(b).FullCell(a,b)$, **support**: {a}, **suppsym**: {}

State S_2**:** $a(b).FullCell(a,b) + \overline{a}a.FullCell(a,a)$, **support**: {a}, **suppsym**: {}

State S_3**:** $a(b).FullCell(a,b) + \overline{a}c.FullCell(a,c)$, **support**: {ac}, **suppsym**: {}

Chapter 3

A modal logic for HD-automata

3.1 A modal logic for the π-calculus: $\mathcal{F}$

We present a very simple Hennessy-Milner logic, which was introduced by Milner, Parrow and Walker [17] and was shown to characterize π-calculus early bisimulation; in the following sections, we will give an interpretation over HD-automata which is adequate for HD-bisimulation, sound and complete w.r.t. the π-calculus interpretation, and suitable for finite-state model checking.

$\mathcal{F}$ is a *temporal* logic, which means that, in addition to usual conjunction and negation constructs, we have one or more *modalities*, which describe properties of a system as it evolves in time (which are called *temporal* properties). In our logic we have modal operators that represent τ, free input, free output and bound output transitions, so we will be able to reason about the behavior of an agent which performs these actions. Typically, to express interesting properties (e.g. *liveness* and *safety* properties) of a concurrent program, we need advanced modal operators like "*eventually*", to state that some property eventually holds after an unspecified number of transitions, or "never". The *eventually* operator, that together with negation can derive the *never* operator, will be included in an extension of $\mathcal{F}$ (sec. 3.6). A powerful modal logic for the π-calculus, with model checking over the version of HD-automata without symmetries, can be found in [8] and in [5], while a very general logic based on fixpoint, called "*name passing μ-calculus*" is presented in [3].

The syntax of $\mathcal{F}$ is:

Definition 3.1 ($\mathcal{F}$ logic).

$$A ::= \bigwedge_{i\in I} A_i \;\Big|\; \neg A \;\Big|\; \langle\alpha\rangle\, A$$

$$\alpha ::= \langle\tau\rangle \;\Big|\; xy \;\Big|\; \overline{x}y \;\Big|\; \overline{x}(y)$$

Note that the logic does not include disjunction, as it can be derived from conjunction and negation by applying De Morgan's laws:

$$A \vee B = \neg(\neg A \wedge \neg B)$$

The constant *true* can be obtained as $\bigwedge_{\emptyset}$ (this will be clearer after giving the satisfaction relation), and $false$ can be obtained as $\neg true$.

The syntax $\langle\alpha\rangle A$ represents modality: the formula holds for a π-calculus agent if, after performing the action α, A holds for the resulting agent. This corresponds to ask for *existence* of a transition matching α. To require that *for all* transitions going out of a state a formula holds, we can use negation again: the formula $\neg\langle\alpha\rangle\neg A$ is true if, for all transitions matching α, the formula A holds in the destination state.

The satisfaction relation is given on a generic π-calculus agent P:

Definition 3.2 (interpreting $\mathcal{F}$ over π-calculus).

$$\begin{array}{lcll}
P & \vDash & \bigwedge_{i\in I} A_i & \Leftrightarrow \quad \forall i \in I. P \vDash A_i \\
P & \vDash & \neg A & \Leftrightarrow \quad P \nvDash A \\
P & \vDash & \langle\alpha\rangle A, \alpha \in \{\tau, xy, \overline{x}y\} & \Leftrightarrow \quad \exists P'. P \xrightarrow{\alpha} P', P' \vDash A \\
P & \vDash & \langle\overline{x}(y)\rangle A & \Leftrightarrow \quad \exists \omega \notin (fn(A) \setminus \{y\}). \exists P'. P \xrightarrow{\overline{x}(\omega)} P', \\
 & & & \qquad P' \vDash A\{\omega/y\}
\end{array}$$

It is important to notice how fresh names are handled: since the name y is bound in $\langle\overline{x}(y)\rangle A$, the choice of y is arbitrary, and we can apply α-conversion to the formula: for an agent P to satisfy the formula $\langle\overline{x}(y)\rangle A$ it is required that $P \xrightarrow{\overline{x}(z)} P'$ and that $\langle\overline{x}(y)\rangle A$ be α-convertible to $\langle\overline{x}(z)\rangle A\{z/y\}$.

3.2 Interpreting $\mathcal{F}$ over HD-automata

We give an interpretation of $\mathcal{F}$ over HD-automata which meets the following goals:

– It's adequate (sec. 3.3) for HD-bisimulation, so that it may become usable, in a generalized definition, not only for HD-automata that are translations of π-calculus agents, but for every HD-automata.

- It is sound and complete w.r.t. the one for π-calculus, so that it can be used as a model checking tool for π-calculus agents.
- It has no side conditions in the definitions for modalities, thanks to the powerful semantics of HD-automata.
- It is suitable for finite-state model checking, so that an algorithmic procedure, which is decidable for finite-state HD-automata, exists.

To define the interpretation of $\mathcal{F}$ formulae over HD-automata, we have to take in account that names in HD states only have a *local* meaning, and not a global one, so we cannot just give a binary satisfaction relation over states and formulae, as in the case of π-calculus, but we have to give a **ternary satisfaction relation** which includes a mapping from the names of a HD state to the names of a formula. We will denote this relation with $\models_{HD}$.

Now we can define the satisfaction relation over HD-states, name mappings and formulae; we will use the more readable notation $q \models^{\rho} A$, where q is a HD-automata state, ρ is a permutation which describes how names in q are to be interpreted into A, and A ranges over $\mathcal{F}$ formulae, instead of $q \models^{\rho}_{HD} A$.

Especially if taking in account lemma A.3, one might wonder if we really need to use such a mapping, or if it would be simpler to say that

$$q \models^{\rho} A \Leftrightarrow q \models^{id} A\rho^{-1}$$

and define a binary relation which always uses the identity as mapping. In the implementation of the model checking algorithm, we will use not only states, but also formulae in normal form. In this case, the mapping becomes necessary since an arbitrary permutation can send a normalized formula into a non normalized one, thus using the normalizing permutation as the mapping in the ternary relation is required.

3.2.1 Design choices

Before giving the formal definition, we will consider its motivations; cases for conjunction and negation in the definition are easy to understand: those just follow an inductive pattern as in the interpretation over π-calculus. The case for modalities is driven by the definitions of bisimulation and of HD-automata corresponding to a π-calculus agent (sec. 2.4). If we want agent q to satisfy formula $\langle\alpha\rangle A$, via the name mapping ρ from its names to those in the formula, it is necessary that:

– q perform a transition, $q \overset{\beta}{\underset{\zeta}{\longmapsto}} q'$, where β has for q, via the permutation ρ, the same *meaning* than α for the formula; for this we will use the function l described above

– q' satisfy the formula A via a suitable permutation of names

Before showing how to choose the name permutations in the mapping for β and q', let's look at some example which will show how to take in account the symmetries of q; in all the following diagrams, arrows represent name mappings, we assume $\sigma \in sym(q)$, and we consider the state q_1 which, as a π-calculus agent, is in the orbit of Q, with $q_1 = \sigma\,(q)$.

For each example, we have to choose a suitable mapping to give meanings to local names of the label and destination state of a transition, in a way that allows the destination state to satisfy the immediate subformula of a modality.

Tau modality

In this case things seem simple:

$$\begin{array}{ccc} q & \xrightarrow{\rho} & \langle\tau\rangle\, A \\ \uparrow{\scriptstyle \zeta} & & \uparrow{\scriptstyle id} \\ q' & \xrightarrow{\xi} & A \end{array}$$

To make the diagram commute, the permutation ξ might be chosen as $(\zeta;\rho)$, so the definition might be:

$$q \vDash^{\rho} \langle\tau\rangle\, A \Leftrightarrow \exists q', \zeta . q \overset{tau}{\underset{\zeta}{\longmapsto}} q' \wedge q' \vDash^{(\zeta;\rho)} A$$

however we will see that also symmetries of states play an important role, because of the other modalities.

Free input and free output modalities

Consider the following statement, which we would like to hold, because it holds for the corresponding π-calculus agent:

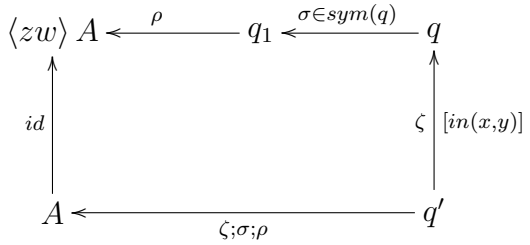
$$a(b).\mathbf{0} \vDash^{id} \langle ac\rangle\, true$$

When we decided (sec. 2.4) to use only representative transitions in the translation from π-calculus agents to HD-automata, we knew that we were representing *all* of the π-calculus agent transitions within a single HD-automata one, by putting symmetries into HD-automata states. On the other hand, we cannot hope that *every* mapping in $sym(q)$ will fit, because there is, for example, the permutation $\{c/d, d/c\}$ which will not map b into c. But we know that there *exists* a permutation $\sigma \in sym(q)$ such that b is mapped into c (this will be exploited in the completeness proof).

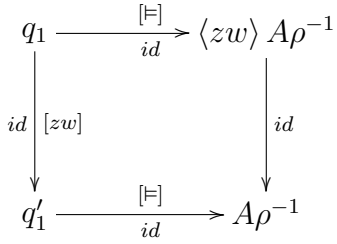
Generally speaking, we need to find a permutation $\sigma \in sym(q)$ such that free names in the modality are mapped to free names in the transition, via ρ, which means that, thanks to symmetries in states, we are allowed to look for a matching transition not only for q , but for any state in the orbit of q.

In the following example diagram, $q \overset{in(x,y)}{\underset{\zeta}{\longmapsto}} q'$ is a HD transition, and names of q_1, which is in the orbit of q, match names in the formula by the permutation ρ, which we assume to be the identity on z and w to ease explanation.

The π-calculus agent q_1 can perform the transition zw (drawn separately), thus it satisfies $\langle zw \rangle A\rho^{-1}$ if $q_1' \vDash A\rho^{-1}$. Moreover we assume that the permutation which exchanges x with z and y with w is in the symmetry of q. Labels of arrows correspond to name substitutions (unless in square brackets, which indicate transition labels or satisfaction relations).



HD automaton



π-calculus agent

This interpretation is particularly well-suited for soundness, because of the structural congruence axiom in the operational semantics of the π-calculus:

$$\frac{A \equiv B \wedge C \equiv D \wedge A \xrightarrow{\alpha} C}{B \xrightarrow{\alpha} D}$$

In the example diagram, $q \equiv q_1$ by definition of HD_π, so the structural congruence axiom has an instance in

$$\frac{q \equiv q_1 \wedge q_1' \equiv q_1' \wedge q_1 \xrightarrow{zw} q_1'}{q \xrightarrow{zw} q_1'}$$

Moreover, we can always find a permutation $\sigma \in sym(q)$ (the identity in the worst case) to map names when interpreting the *tau* modality, so the requirement of existence for the permutation σ does not affect our previous definition; the new one can be given as:

$$\begin{aligned} & q \vDash^{\rho} \langle \alpha \rangle A, \alpha \in (\tau, xy, \overline{x}y) \\ \Leftrightarrow \quad & \exists \sigma \in sym(q). \exists q', \zeta \; . \; q \xrightarrow[\zeta]{\left(\rho^{-1};\sigma^{-1}\right)(l(\alpha))} q' \wedge q' \vDash^{(\zeta;\sigma;\rho)} A \end{aligned}$$

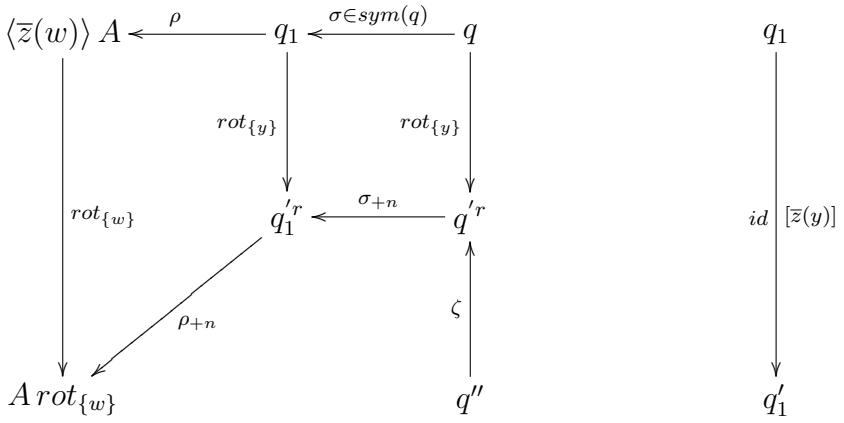
The same considerations hold for the free output modality, and the same definition can be used.

Bound output modality

Now we have to take into account the generation of fresh names. As for the other modalities, free names in the label can correspond to free names in the modal operator by $(\sigma;\rho)$ where $\sigma \in sym(q)$, but to be sure that the generated name is fresh in the formula, and corresponds to the generated name in the HD transition, given the formula $\langle \overline{x}(y) \rangle A$, when recurring in the definition, we apply the substitution $rot_{\{y\}}$ to A, so that the fresh name is always x_0 both in the destination state of the transition and in the immediate subformula of the modal formula.

The following diagram shows how names are mapped in the case of a transition $q \xrightarrow[\zeta]{bout(x)} q''$. The agent q_1, which is in the orbit of q, can perform the transition $q_1 \xrightarrow{\overline{z}(w)} q_1'$, drawn in square brackets, and $\{x/z, z/x\}$ is in its symmetry. According to the definition of HD_{π} (sec. 2.4), names of q are rotated, to send the newly generated name into x_0; this is represented with the states $q_1'^{r}$ and q'^{r}; names in $\langle \overline{z}(w) \rangle A$ are rotated accordingly, and for ease of readability we assume that $\rho(z) = z$. The number n is the number of fresh names generated along the transition (for *bout* transitions, it's 1).

Under these conditions, it's clear that $q_1 \vDash \langle \overline{z}(w) \rangle A\rho^{-1}$ if $q_1' \vDash A\rho^{-1}\{a/w\}$, where $a \notin fn(A)$, and so we want q to satisfy $\langle \overline{z}(w) \rangle A$ via ρ.



HD automaton π-calculus agent

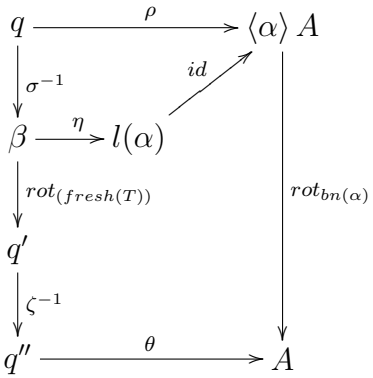
As displayed in the diagram, names in q'' have to correspond to names in A by $(\zeta; \sigma_{+n}; \rho_{+n})$; this is not a trouble for previous definitions, since if no fresh name is generated, the number n is equal to zero, and the rotation is over an empty set and becomes the identity substitution.

3.2.2 Formal definition

The above examples explain how we can make design choices for the satisfaction relation of $\mathcal{F}$ over HD-automata, keeping in mind soundness and completeness w.r.t. the interpretation over π-calculus; now we will give a formal definition, slightly more general than the ones we have just seen (e.g. it potentially allows labels generating more than one fresh name).

For a HD-automata state q to satisfy a formula $\langle\alpha\rangle A$ via the mapping ρ it is required that:

- a transition $T = q \overset{\beta}{\underset{\zeta}{\longmapsto}} q'$ exist
- α correspond to β by l and by a suitable mapping η over free names
- the number n of fresh names generated by T be the same of the bound names in α
- q' satisfy A by appropriate name substitutions
- the following diagram commute, where $\sigma \in sym(q)$ and $fresh(T)$ denotes those bound names in q that are instantiated with fresh names by T



The only unknowns in the diagram are η and θ. The diagram commutes with $\eta = \sigma;\rho$ and $\theta = \zeta;(\sigma;\rho)_{+n}$, where n is the number of bound names in α, as we will show in the adequacy proof.

Definition 3.3 (Interpreting $\mathcal{F}$ over HD-automata).

$$
\begin{array}{lcl}
q \models^{\rho} \bigwedge_{i\in I} A_i & \Leftrightarrow & \forall i \in I. q \models^{\rho} A_i \\
q \models^{\rho} \neg A & \Leftrightarrow & q \not\models^{\rho} A \\
q \models^{\rho} \langle\alpha\rangle A & \Leftrightarrow & \exists\, (\sigma \in sym(q), q', \zeta)\, .q \xrightarrow[\zeta]{(\rho^{-1};\sigma^{-1})(l(\alpha))} q', \\
 & & q' \models^{(\zeta;(\sigma;\rho)_{+n})} A\, rot_{bn(\alpha)}, \\
 & & \text{where } \alpha \text{ has } n \text{ bound names}
\end{array}
$$

3.2.3 Example: checking some property of a buffer

Now we can check some property for the buffer of our previous examples:

$$Buf(c) \quad \stackrel{\text{def}}{=} \quad c(x).\bar{c}x.Buf(c)$$

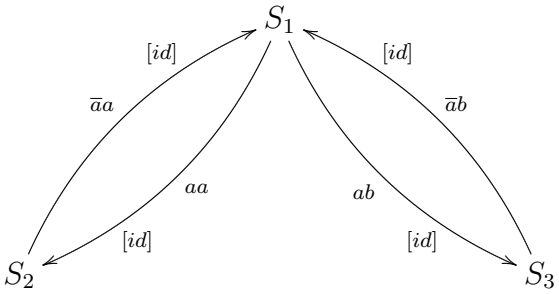
we can require that every time a value is stored into the buffer, it is immediately available for retrieving:

$$mem \quad = \quad \langle ca\rangle\,\langle\bar{c}a\rangle\, true$$

also, we can check that it's impossible to retrieve a value when we just have stored another one:

$$consistent \quad = \quad \neg(\langle ca\rangle\,\langle \overline{c}b\rangle\, true)$$

By looking at the HD-automaton



State S_1**:** $a(b)\overline{a}b.Buf(a)$, **support**: {a}, **suppsym**: {}

State S_2**:** $\overline{a}a.Buf(a)$, **support**: {a}, **suppsym**: {}

State S_3**:** $\overline{a}b.Buf(a)$, **support**: {a,b}, **suppsym**: {}

we see that $S_1 \vDash^{\rho} mem$, where $\rho = \{a/c, c/a\}$, i.e. the permutation that normalizes $Buf(c)$ into S_1; for ease of explanation we check the equivalent assertion

$$S_1 \vDash^{id} mem\,\rho^{-1} = \langle ab\rangle\,\langle \overline{a}b\rangle\, true$$

There is a transition labeled ab from S_1 to S_2, and then one labeled $\overline{a}b$ from S_2 to S_1, so the formula is true without having to use any permutation σ.

Now we look at

$$S_1 \vDash^{id} consistent\,\rho = \neg(\langle ac\rangle\,\langle \overline{a}b\rangle\, true)$$

We notice that we don't have a transition labeled ac leaving S_1, but there is ab, and permutation $\{b/c, c/b\}$ is in the symmetry of S_1 (α-conversion), so we apply the name mapping id as prescribed in the definition, and check if $S_2 \vDash^{id} \langle \overline{a}c\rangle\, true$. This is false, because we don't have any transition labeled $\overline{a}c$, but only $\overline{a}b$, and $\{b/c, c/b\}$ is not in the symmetry of S_2.

3.3 Adequacy

In a modal logic $\mathcal{L}$ interpreted over an ordinary labeled transition system (S, L, T), we say that $\mathcal{L}$ is *adequate* w.r.t. the relation $\mathcal{B} \subseteq S \times S$ (which is usually the bisimilarity relation) if

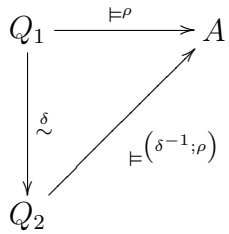
$$(\forall F \in \mathcal{L}.\forall A, B \in S.A \vDash F \Leftrightarrow B \vDash F) \Leftrightarrow \langle A, B\rangle \in \mathcal{B}$$

that is, A is bisimilar to B if and only if A and B satisfy the same formulae. A logic is also said to *characterize* $\mathcal{B}$ if this property holds.

Since names for HD-automata are local, the definition of bisimulation takes into account name mappings (sec. 2.3), and the definition of adequacy has to be restated accordingly: we say that a logic $\mathcal{L}$ is adequate w.r.t. HD-bisimulation if for all states Q_1 and Q_2 we have

$$Q_1 \overset{\delta}{\sim} Q_2 \Leftrightarrow \left(\forall A \in \mathcal{L}.Q_1 \vDash^{\rho} A \Leftrightarrow Q_2 \vDash^{(\delta^{-1};\rho)} A\right)$$

The name mapping $(\delta^{-1};\rho)$ can be understood with the help of the following diagram (remember that δ is invertible):



In the adequacy proof, we will exploit the obvious fact that

$$Q_1 \overset{\delta}{\sim} Q_2 \Leftrightarrow \exists R.R \text{ is a HD-bisimulation } \wedge \langle Q_1, \delta, Q_2\rangle \in R$$

Adequacy of $\mathcal{F}$

The proof that $\mathcal{F}$ is adequate for HD-bisimilarity is split into the "if" and the "only if" parts.

To show the former we prove that the relation

$$S = \left\{\langle Q_1, \delta, Q_2\rangle \,\middle|\, \forall A \in \mathcal{F}.Q_1 \vDash^{\rho} A \Leftrightarrow Q_2 \vDash^{(\delta^{-1};\rho)} A\right\}$$

is a HD-bisimulation.

The latter can be shown by induction over the rules generating the formulae in $\mathcal{F}$: we prove that for HD-automata Q_1 and Q_2 bisimilar via permutation δ, and for *any* formula $A \in \mathcal{F}$, the following property holds:

$$P(A) = Q_1 \vDash^{\rho} A \Leftrightarrow Q_2 \vDash^{(\delta^{-1};\rho)} A$$

The formal proof follows (note that we use the shorthand $\forall$ (or $\exists$) $Q \overset{\mu}{\underset{\zeta}{\longmapsto}} Q'$ for $\forall$ (or $\exists$) Q, Q', μ, ζ (such that) $Q \overset{\mu}{\underset{\zeta}{\longmapsto}} Q'$).

Theorem 3.4. $\mathcal{F}$ is adequate for HD-bisimulation.

"if" part:

$$\left(\forall A \in \mathcal{F}.Q_1 \vDash^{\rho} A \Leftrightarrow Q_2 \vDash^{\left(\delta^{-1};\rho\right)} A\right) \Rightarrow Q_1 \overset{\delta}{\sim} Q_2$$

Proof. Let $S = \left\{\langle Q_1, \delta, Q_2\rangle \,\middle|\, \forall A \in \mathcal{F}.Q_1 \vDash^{\rho} A \Leftrightarrow Q_2 \vDash^{\left(\delta^{-1};\rho\right)} A\right\}$, and suppose that S is not a bisimulation. Be $\langle Q_1, \delta, Q_2\rangle \in S$.

S is not a bisimulation

$\Rightarrow$ $\exists \sigma_1 \in sym(Q_1).\exists Q_1 \overset{l_1}{\underset{\zeta_1}{\longmapsto}} Q_1'.\forall \sigma_2 \in sym(Q_2).\forall Q_2 \overset{l_2}{\underset{\zeta_2}{\longmapsto}} Q_2'.$

$l_2 \neq \gamma\left(l_1\right) \vee \langle Q_1', \delta', Q_2'\rangle \notin S$

where $\gamma = \left(\sigma_1; \delta; \sigma_2^{-1}\right)$, n is the number of bound names in l and

$\delta' = \left(\zeta_1; \gamma_{+n}; \zeta_2^{-1}\right)$

Now we can analyze the two cases separately, using definition A.6 to build a modal formula from a transition:

1. No l_2 matches l_1:

   $\forall \sigma_2 \in sym(Q_2).\forall Q_2 \overset{l_2}{\underset{\zeta_2}{\longmapsto}} Q_2'.l_2 \neq \gamma\left(l_1\right)$

   $\equiv$ $\neg\exists\left(\sigma_2 \in sym(Q_2), Q_2', l_2, \zeta_2\right)\ .\ Q_2 \overset{\left(\delta^{-1};\sigma_1^{-1}\right)^{-1};\sigma_2^{-1}\ \left(l_1\right)}{\underset{\zeta_2}{\longmapsto}} Q_2'$

   $\Rightarrow$ $\left\{\text{definition of } \vDash_{HD} \text{ (3.3) with } \rho = \left(\delta^{-1}; \sigma_1^{-1}\right)\ \right\}$

   $Q_1 \vDash^{\sigma_1^{-1}} \langle m(l_1)\rangle\, true \wedge Q_2 \nvDash^{\left(\delta^{-1};\sigma_1^{-1}\right)} \langle m(l_1)\rangle\, true$

   and this clearly contradicts the hypothesis. Note that $Q_1 \vDash^{id} \langle m(l_1)\rangle$, but since $\sigma_1 \in sym(Q_1)$ we also have $Q_1 \vDash^{\sigma_1^{-1}} \langle m(l_1)\rangle$.

2. No Q'_2 matches Q'_1

   $\langle Q'_1, \delta', Q'_2 \rangle \notin S$

   $\Rightarrow$ {definition of S}

   $\exists \bar{\rho}. \exists \bar{f} \in \mathcal{F}. Q'_1 \vDash^{\bar{\rho}} \bar{f} \wedge Q'_2 \nvDash^{\left(\delta'^{-1};\bar{\rho}\right)} \bar{f}$

   $\Rightarrow$ {lemma $A.3$}

   $\forall \rho \text{ injective } . \exists f^\rho = \bar{f}\left(\bar{\rho}^{-1}; \rho\right). Q'_1 \vDash^{\rho} f^\rho \wedge Q'_2 \nvDash^{\left(\delta^{-1};\rho\right)} f^\rho$

   Using the function m^* defined at A.6 we will now build from f^ρ a modal formula A such that $Q_1 \vDash^{id} A$ but $Q_2 \nvDash^{\delta^{-1}} A$, which contradicts the hypothesis and completes the proof.

   Be n the number of fresh names generated by l_1 (and l_2), and $\mu = \sigma_1(m^*_{x_i}(l_1))$, where $x_{i+n} = min\left(\mathfrak{N} \setminus fn\left(f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}\right)\right)$.

   Be $\gamma(x) = y$ iff $rot_{bn(\mu)}(y) = x$ (i.e. γ is the inverse of $rot_{bn(\mu)}$ restricted to its image). Notice that $\gamma; rot_{bn(\mu)} = id$.

   Then $Q_1 \vDash^{id} \langle \mu \rangle f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}\gamma$: according to the definition of $\vDash_{HD}$ we have

   $$\sigma_1 \in sym(Q_1) \wedge Q_1 \xrightarrow[\zeta_1]{\left(id^{-1};\sigma_1^{-1}\right)(l(\mu)))} Q'_1$$

   by definition of m^*,l and μ, and

   $$Q'_1 \vDash^{\left(\zeta_1;(\sigma_1;id)_{+n}\right)} f^{\left(\zeta_1;(\sigma_1)_{+n}\right)} \gamma\, rot_{bn(\mu)}$$

   which is the same as

   $$Q'_1 \vDash^{\left(\zeta_1;(\sigma_1)_{+n}\right)} f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}$$

   by construction of f^ρ.

   Now let's suppose that Q_2 satisfy the same formula via δ^{-1}:

   $Q_2 \vDash^{\delta^{-1}} \langle \mu \rangle f^{\zeta_1;\left((\sigma_1)_{+n}\right)\gamma}$

   $\Rightarrow$ $\exists \sigma_2 \in sym(Q_2), l_2, \zeta_2. Q_2 \xrightarrow[\zeta_2]{\delta;\sigma_2^{-1}(l(\mu))} Q'_2 \wedge$

   $Q'_2 \vDash {}^{\zeta_2;\left(\sigma_2;\delta^{-1}\right)_{+n}} \quad f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}$

The first part of the conjunction, by definition of μ, brings in scope the same set of states Q_2' which we are supposing not bisimilar to Q_1' via δ', so we have a contradiction from the second part of the conjunction; let's start with the definition of f^ρ:

$$Q_2' \not\models^{\left((\delta')^{-1};\zeta_1;(\sigma_1)_{+n}\right)} f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}$$

$\Rightarrow$ {definition of δ'}

$$Q_2' \not\models {}^{\zeta_2;\ \sigma_2;\delta^{-1};\left(\sigma_1^{-1}\right)_{+n}\ ;\zeta_1^{-1}\ ;\zeta_1;(\sigma_1)_{+n}} f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}$$

$\Rightarrow$ {properties of substitutions}

$$Q_2' \not\models {}^{\zeta_2;\left(\sigma_2;\delta^{-1}\right)_{+n}} f^{\left(\zeta_1;(\sigma_1)_{+n}\right)}$$

This contradicts the previous result.

□

"only if" part:

$$Q_1 \overset{\delta}{\sim} Q_2 \Rightarrow \left(Q_1 \models^\rho A \Leftrightarrow Q_2 \models^{\left(\delta^{-1};\rho\right)} A\right)$$

proof (by induction over the structure of A).

1. Conjunction:

   $$Q_1 \models^\rho \bigwedge_{i\in I} A_i$$

   $\Leftrightarrow$ $\forall i \in I. Q_1 \models^\rho A_i$

   $\Leftrightarrow$ {induction hypothesis}

   $$\forall i \in I. Q_2 \models^{\left(\delta^{-1};\rho\right)} A_i$$

   $\Leftrightarrow$ $Q_2 \models^{\left(\delta^{-1};\rho\right)} \bigwedge_{i\in I} A_i$

2. Negation:

$Q_1 \models^{\rho} \neg A$

$\Leftrightarrow$ $Q_1 \not\models^{\rho} A$

$\Leftrightarrow$ {induction hypothesis}

$Q_2 \not\models^{\left(\delta^{-1};\rho\right)} A$

$\Leftrightarrow$ $Q_2 \models^{\left(\delta^{-1};\rho\right)} \neg A$

3. Modalities:

$Q_1 \models^{\rho} \langle\alpha\rangle A,$ where α has n bound names

$\Leftrightarrow$ $\exists\left(\sigma_1 \in sym(Q_1), Q_1', \zeta_1\right).Q_1 \xrightarrow[\zeta_1]{\left(\rho^{-1};\sigma_1^{-1}\right)(l(\alpha))} Q_1' \wedge$

$Q_1' \models^{\left(\zeta_1;(\sigma_1;\rho)_{+n}\right)} A\, rot_{bn(\alpha)}$

$\Leftrightarrow$ $\left\{Q_1 \overset{\delta}{\sim} Q_2\right\}$

$\exists\sigma_2 \in sym(Q_2).Q_2 \xrightarrow[\zeta_2]{\left(\left(\rho^{-1};\sigma_1^{-1}\right);\left(\sigma_1;\delta;\sigma_2^{-1}\right)\right)(l(\alpha))} Q_2' \wedge$

$Q_1' \overset{\zeta_1;\left(\sigma_1;\delta;\sigma_2^{-1}\right)_{+n};\zeta_2^{-1}}{\sim} Q_2'$

$\Leftrightarrow$ {induction hypothesis, properties of substitutions}

$\exists\sigma_2 \in sym(Q_2).Q_2 \xrightarrow[\zeta_2]{\left(\rho^{-1};\delta;\sigma_2^{-1}\right)(l(\alpha))} Q_2' \wedge$

$Q_2' \models^{\zeta_2;\left(\sigma_2;\delta^{-1};\sigma_1^{-1}\right)_{+n};\zeta_1^{-1};\zeta_1;(\sigma_1;\rho)_{+n}} A\, rot_{bn(\alpha)}$

$\Leftrightarrow$ {properties of substitutions}

$\exists\sigma_2 \in sym(Q_2).Q_2 \xrightarrow[\zeta_2]{\left(\delta^{-1};\rho\right)^{-1};\sigma_2^{-1}\ \ (l(\alpha))} Q_2' \wedge$

$Q_2' \models^{\zeta_2;\left(\sigma_2;\left(\delta^{-1};\rho\right)\right)_{+n}} A\, rot_{bn(\alpha)}$

$\Leftrightarrow \; Q_2 \vDash^{\left(\delta^{-1};\rho\right)} \langle\alpha\rangle A$

□

3.4 Soundness and completeness

We already know that $\mathcal{F}$ is adequate for HD-automata bisimulation, that it's adequate for π-calculus early bisimulation [17], and that the translation HD_π respects bisimulation [19], so $(P \sim_e Q) \wedge (P \vDash A) \Rightarrow Q \vDash A \wedge HD_\pi(P) \overset{\delta}{\sim} HD_\pi(Q)$ for some δ. The only thing that is still missing is to show that P and $HD_\pi(P)$ satisfy the *same* formulae (with correct name mappings in the HD satisfaction relation); this proof of soundness and completeness can be done by induction over the rules generating $\mathcal{F}$.

Theorem 3.5 (Soundness and completeness).

$$\bar{P} \vDash A, norm(\bar{P}) = \langle P, \rho\rangle \Leftrightarrow HD_\pi(P) \vDash^\rho A$$

Proof (by induction over the rules generating $\mathcal{F}$). Cases for negation and conjunction are trivial, while the proof for modalities uses the following observation:

Observation 3.6. If $P \xrightarrow{\alpha} P'$ is a π-calculus transition, there always exists $\sigma \in sym(P)$ such that $P \xrightarrow{\sigma(\alpha)} P'\sigma$ is a representative transition (the set of transitions corresponding to P is non empty, and the set of labels is well founded so it has a minimum element). We also have $P\sigma \xrightarrow{\sigma(\alpha)} P'\sigma$ because of the structural congruence axiom.

– Tau, free input and free output modalities:

$\bar{P} \vDash \langle\alpha\rangle A, norm(\bar{P}) = \langle P, \rho\rangle, \alpha \neq \overline{x}(y)$

$\Leftrightarrow$ {Lemma A.2}

$P \vDash (\langle\alpha\rangle A)\,\rho^{-1}$

$\Leftrightarrow \; P \vDash \langle\rho^{-1}(\alpha)\rangle A\rho^{-1}$

$\Leftrightarrow \; \exists P'.P \xrightarrow{\rho^{-1}(\alpha)} P' \wedge P' \vDash A\rho^{-1}$

$\Leftrightarrow$ {Lemma A.2, observation 3.6}

$\exists P', \sigma^{-1} \in sym(P).P\sigma^{-1} \xrightarrow{(\rho^{-1};\sigma^{-1})(\alpha)} P'\sigma^{-1}$ is a representative

transition $\wedge\, P'\sigma^{-1} \vDash A\,(\rho^{-1};\sigma^{-1})$

$\Leftrightarrow$ $\{Q = P\sigma^{-1}, Q' = P'\sigma^{-1}\}$

$\exists \sigma \in sym(Q), Q'.Q \xrightarrow{(\rho^{-1};\sigma^{-1})(\alpha)} Q'$ is a representative transition $\wedge$

$Q' \vDash A\,(\rho^{-1};\sigma^{-1})$

Since the π-calculus transition is representative, we can obtain the HD state corresponding to Q and the HD transition from Q to Q'; then, by induction hypothesis, we will also transform the satisfaction of A from a property of the π-calculus agent Q' to a property of the corresponding HD state:

$\{norm(Q') = \langle Q'', \zeta\rangle\}$

$\exists \sigma \in sym(Q), Q', \zeta.Q \xrightarrow[\zeta]{(\rho^{-1};\sigma^{-1})(l(\alpha))} Q'' \wedge Q'' \vDash A\,(\rho^{-1};\sigma^{-1};\zeta^{-1})$

$\Leftrightarrow$ {induction hypothesis, with $norm(Q'') = \langle Q'', id\rangle$}

$\exists \sigma \in sym(Q), Q', \zeta.Q \xrightarrow[\zeta]{(\rho^{-1};\sigma^{-1})(l(\alpha))} Q'' \wedge Q'' \vDash^{id} A\,(\rho^{-1};\sigma^{-1};\zeta^{-1})$

$\Leftrightarrow$ {Lemma A.3}

$\exists \sigma \in sym(Q), Q', \zeta.Q \xrightarrow[\zeta]{(\rho^{-1};\sigma^{-1})(l(\alpha))} Q'' \wedge Q'' \vDash^{(\zeta;\sigma;\rho)} A$

$\Leftrightarrow$ {Definition of $\vDash_{HD}$}

$Q \vDash^{\rho} \langle\alpha\rangle\, A$

– Bound output modality

The proof for the bound output modality requires particular care and is split into the *if* and the *only if* parts. The reader should remind that for a π-calculus agent P it holds $\sigma \in sym(P) \Rightarrow \sigma^{-1} \in sym(P)$.

1. From agents to states ($\Rightarrow$):

$$\bar{P} \vDash \langle \overline{x}(y) \rangle A, norm(\bar{P}) = \langle P, \rho \rangle$$

$$\Rightarrow \; P \vDash (\langle \overline{x}(y) \rangle A)\, \rho^{-1}$$

$$\Rightarrow \; P \vDash \langle \rho^{-1}(\overline{x}(y)) \rangle A\rho^{-1}$$

$$\Rightarrow \; \{\text{definition of } \vDash\}$$

$$\exists \omega \notin fn(A\rho^{-1}).\exists P'.P \xrightarrow{\rho^{-1}(\overline{x}(w))} P' \wedge P' \vDash A\,(\rho^{-1}; \{w/\rho^{-1}(y)\})$$

$$\Rightarrow \; \{\text{Observation 3.6}\}$$

$$\exists \omega \notin fn(A\rho^{-1}), \sigma^{-1} \in sym(P), P'.P \xrightarrow{\left(\rho^{-1};\sigma^{-1}\right)(\overline{x}(z))} P'\sigma^{-1}$$

$$\text{is representative } \wedge P'\sigma^{-1} \vDash A\,(\rho^{-1}; \{w/\rho^{-1}(y)\}\,; \sigma^{-1})$$

$$\Rightarrow \; \left\{norm\left(P'\sigma^{-1} rot_{\{\sigma^{-1}(w)\}}\right) = \langle P'', \zeta \rangle\,, \text{definition of } HD_\pi\right\}$$

$$\exists \omega \notin fn(A\rho^{-1}), P'', \zeta, \sigma \in sym(P).P \xrightarrow[\zeta]{\rho^{-1};\sigma^{-1}(bout(x))} P'' \wedge$$

$$P'\left(\sigma^{-1}; rot_{\sigma^{-1}(w)}\right) \vDash A\left(\rho^{-1}; \{w/\rho^{-1}(y)\}\,; \sigma^{-1}; rot_{\{\sigma^{-1}(w)\}}\right)$$

$$\Rightarrow \; \left\{\text{induction hypothesis}, \gamma = \left(\rho^{-1}; \{w/\rho^{-1}(y)\}\,; \sigma^{-1}; rot_{\{\sigma^{-1}(w)\}}\right)\right\}$$

$$\exists P'', \zeta.P \xrightarrow[\zeta]{\rho^{-1};\sigma^{-1}(bout(x))} P'' \wedge P'' \vDash^{\zeta} A\gamma$$

Now, if we show that γ, restricted to free names in A, is equal to $\left(rot_y; (\rho^{-1}; \sigma^{-1})_{+1}\right)$, the thesis is proved, since, for σ bijective, $P \vDash^{\alpha} A\sigma \Leftrightarrow P \vDash^{\alpha;\sigma^{-1}} A$ (lemma A.3), and so we would have:

$$\exists \sigma \in sym(P).P \xrightarrow[\zeta]{\rho^{-1};\sigma^{-1}(l(\overline{x}(y)))} P'' \wedge P'' \vDash^{\left(\zeta;(\sigma;\rho)_{+1}\right)} A\, rot_y.$$

It's easy to verify this fact: suppose $x_i \in fn(A)$; if $x_i = y$, then $\gamma(x_i) = x_0$, else $\gamma(x_i) = y_{i+1}$, where $y_i = (\rho^{-1}; \sigma^{-1})\,(x_i)$; this is by definition $(\rho^{-1}; \sigma^{-1})\,(x_{i+1})$, and since $x_i \neq y$ we know that $rot_y(x_i) = x_{i+1}$, hence the thesis.

2. From states to agents ($\Rightarrow$):

$$Q \vDash^{\rho} \langle \overline{x}(y) \rangle A$$

$$\Rightarrow \ \exists \sigma \in sym(Q).Q \xrightarrow[\zeta]{bout\left(\left(\rho^{-1};\sigma^{-1}\right)(x)\right)} Q'' \wedge$$

$$Q'' \vDash^{\left(\zeta;(\sigma;\rho)_{+1}\right)} A\, rot_y$$

$$\Rightarrow \ \{\text{definition 2.6}\}$$

$$Q \overset{\overline{(\rho^{-1};\sigma^{-1})(x)}(w'')}{\longrightarrow} Q' \wedge norm\left(Q' rot_{w''}\right) = \langle Q'', \zeta \rangle$$

Using α-conversion, we choose w'' such that $(\rho(\sigma(w))) \notin fn\,(A)$, which will be used later in the proof. We have to apply the induction hypothesis over Q'', and then lemma A.2: since rot_s is always injective, by looking at the thesis of this theorem we have:

$$Q' rot_{w''} \vDash A\, rot_y \left(\rho^{-1};\sigma^{-1}\right)_{+1}$$

We introduce the substitution

$$unrot_{(k,i)}(x_j) = \begin{cases} x_{i+j} & \text{if } j \in [0, k-1] \\ x_{j-k} & \text{if } j \in [k, i] \end{cases}$$

which in is the "opposite" of $rot_{\{n_1,\ldots n_k\}}$, in the sense of lemma A.5. Now we consider the π-calculus transition that we have found:

$$Q \overset{\overline{(\rho^{-1};\sigma^{-1})(x)}(w'')}{\longrightarrow} Q'$$

$$\Rightarrow \ \{\text{lemma A.1, } w' = \sigma(w'')\}$$

$$Q\sigma \overset{\overline{(\rho^{-1})(x)}(w')}{\longrightarrow} Q'\sigma$$

$$\Rightarrow \ \{\sigma \in sym(Q), \text{structural congruence rule}\}$$

$$Q \overset{\overline{\rho^{-1}(x)}(w')}{\longrightarrow} Q'\sigma$$

$$\Rightarrow \ \{w = \rho(w'), \text{former result}\}$$

$$Q\rho \xrightarrow{\overline{x}(w)} Q'(\sigma;\rho) \wedge Q' rot_{w''} \vDash A\, rot_y \left(\rho^{-1};\sigma^{-1}\right)_{+1}$$

$$\Rightarrow \ Q\rho \xrightarrow{\overline{x}(w)} Q'(\sigma;\rho) \wedge Q' rot_{w''} \left(\sigma;\rho\right)_{+1} \vDash A\, rot_y$$

$\Rightarrow$ {lemma A.4}

$$Q\rho \xrightarrow{\overline{x}(w)} Q'(\sigma;\rho) \wedge Q'(\sigma;\rho;rot_w) \vDash A\, rot_y$$

$$\Rightarrow\ Q\rho \xrightarrow{\overline{x}(w)} Q'(\sigma;\rho) \wedge Q'(\sigma;\rho);rot_w;unrot_{(1,w)} \vDash A\ \left(rot_y;unrot_{(1,w)}\right)$$

$\Rightarrow$ $\{\text{lemma A.5}, w \notin fn\,(A)\}$

$$\exists w \notin fn\,(A)\,.Q\rho \xrightarrow{\overline{x}(w)} Q'(\sigma;\rho) \wedge Q'(\sigma;\rho) \vDash A\,\{w\!/\!y\}$$

$\Rightarrow$ {definition 3.2}

$$Q\rho \vDash \langle\overline{x}(y)\rangle\, A$$

Since Q is in normal form we have $norm\,(Q\rho) = \langle Q, \rho\rangle$ and this concludes the proof.

□

3.5 The late semantics of the π-calculus

We have seen the *early* semantics of the π-calculus, but there are other interpretations of the syntax; the very first one proposed by Milner, Parrow and Walker was the *late* one, which is tighter in the sense that two late bisimilar processes are early bisimilar, but the converse does not hold. The main point of the late semantics w.r.t. the early one is that, in the input action, the received name is not free but bound. Bisimilarity is changed accordingly: two states which are sources of the transitions $P_1 \xrightarrow{x(y)} P_1'$ and $P_2 \xrightarrow{x(z)} P_2'$ match if P_1' and P_2' are bisimilar for every choice of y that P_2 can be α-converted to, while in the early semantics we can choose a different input transition (e.g. in a choice construct) for every input name; thus the following two processes are early bisimilar but not late bisimilar (the example is due to [17]):

$$x(u).\tau.\mathbf{0} + x(u).\mathbf{0} \sim_e P.\mathbf{0} + x(u).[u = z]\tau.\mathbf{0}$$

3.5.1 Modalities for the late semantics

As shown in [6], the late semantics of the π-calculus can be expressed using two transition steps, which have to be considered atomically; a transition of the form

$$P \xrightarrow{x(y)} P'$$

is turned into

$$P \xrightarrow{x} \lambda y.P' \xrightarrow{[z]} P'\{y/z\}$$

where the second step is infinite branching and the name z is free, similarly to what happens for the free input modality. This method of giving a late semantics has the advantage that the definition of bisimulation can be the *early* one.

The logic $\mathcal{F}$ can be enriched with the two corresponding modalities, which we call respectively $\langle x \rangle$ and $\langle [x] \rangle$.

In [17] a modal logic, adequate for the late semantics of the π-calculus, is presented. This logic is $\mathcal{F}$ extended with bound input $x(y)$ and matching $[x = y]$ constructs. Even if at first sight the approach followed in our work may look very different from the one in [17], these are similar:

- the modality $\langle x \rangle$ can be seen as $\langle x(y) \rangle$, because the name y is bound, so the only information that one gets from the truth of $P \vDash \langle x(y) \rangle \, true$ is that there *exists* a name which has been received on channel x;
- matching is not useful in contexts differing from a bound input modality, because names in other contexts can only be free or fresh, and in both cases the result of matching can be known by just examining the agent syntactically. Here is an example formula without fresh names:

  $$\langle \overline{x} y \rangle \, \langle [x = y] \rangle \, true$$

  We already know that the matching will fail, because x and y are different. An example involving bound output is:

  $$\langle \overline{x}(y) \rangle \, \langle [y = z] \rangle \, true$$

  where, again, we already know that the matching will always fail, since y is different from any other name.

Our operator $\langle [x] \rangle$ corresponds to matching specialized to the *immediately preceding* input transition; this is the reason why it is not as expressive as [17]: the real matching can be postponed at any time or even not done. Consider the following example:

$$P \stackrel{\text{def}}{=} x(y).\overline{x}x.\mathbf{0}$$

With bound input we can assert:

$$P \vDash \langle x(y) \rangle \langle \overline{x}x \rangle \, true$$

while to describe the same property with our version of the late semantics and the corresponding logic, we are compelled to use the second step of the input transition, so we have to match all the possibilities (which are in a finite number, namely one for every free name of the agent, plus one for all the non-free names which are identified in the agent's orbit), and the required formula depends on the agent:

$$P \vDash \langle x \rangle \langle [x] \rangle \langle \overline{x}x \rangle \, true \vee \langle x \rangle \langle [y] \rangle \langle \overline{x}x \rangle \, true$$

3.5.2 Interpretation over HD-automata and adequacy

The interpretation over HD-automata is straightforward: we define the labels *binp(x)*, which announces input on channel x, and *conc(z)*, which corresponds to the second step, often called *concretion*.

A π-calculus transition $P \xrightarrow{x(y)} P'$ is translated into (without considering normalizations for simplicity) $P \xrightarrow{binp(x)} \lambda(y).P' \xrightarrow{conc(y)} P'$, where the y in $\lambda(y)$ is α-convertible as usual, and the symmetry of the intermediate state behaves accordingly. Both names in the labels are free, and there is no fresh name generated along any of the two transitions. Of course, states should be normalized, and only representative transitions should be used, as it is stated in definition 2.6.

The only thing that differs in the interpretation of the logic (definition 3.3) is the function l, which turns modalities of the logic into labels of the HD-automata and has to be changed to reflect the addition of the two new labels.

The adequacy proof remains unchanged, so the new logic is still adequate for HD bisimulation. This is a consequence of the fact that the function l is still bijective modulo α-conversion of labels; for any formalism mapped into HD-automata, which has a modal logic, if we manage to find a bijective l to map this modal logic into the HD one, the adequacy proof holds.

3.6 Extending $\mathcal{F}$: $\mathcal{F}^*$

3.6.1 An *"eventually"* temporal operator

The ability to express *liveness* and *safety* properties is very useful in a modal logic; for this purpose we add to the logic $\mathcal{F}$ a very simple "eventually" operator, and give an interpretation for this new construct over π-calculus and over HD-automata. This modal operator is referred to by the syntax $E\,\Phi$, where Φ is any formula. The logic $\mathcal{F}$ is thus augmented with a new construct; we will call this new logic $\mathcal{F}^*$.

Definition for π-calculus

The definition for π-calculus agents is simple in principle: we just require that agent P either satisfy Φ or perform any transition and then satisfy $E\,\Phi$ in the resulting state. In practice we also need to avoid free name capture in bound output transitions, so we require that, if agent P satisfies Φ after k transitions, bound names of all those transitions don't appear in Φ as free names. This makes sense, since these names should be unknown to the formula Φ.

Definition 3.7 (Interpreting $E\,\Phi$ over π-calculus).

$$P_0 \vDash E\,\Phi \;\Leftrightarrow\; \exists k \geq 0. \exists P_1, \ldots, P_k. P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n$$

$$\wedge P_n \vDash \Phi \wedge \forall i \in [1,k]. bn(k) \cap fn(\Phi) = \emptyset$$

The definition should be a little more careful about what happens when $k = 0$; if $k = 0$ we do not require the existence of any transition, but simply that P_0 satisfy Φ.

Note that the same definition can be given in a recursive way, as we will do for clarity in the case of HD-automata, and the following definition is equivalent to the one we have just seen:

$$P \vDash E\,\Phi \;\Leftrightarrow\; P \vDash \Phi \vee (P \xrightarrow{\alpha} P_1 \wedge bn(\alpha) \cap fn(\Phi) = \emptyset \wedge P_1 \vDash E\,\Phi)$$

Definition for HD-automata

The definition over HD-automata is somewhat complicated by the locality of names in HD states. Luckily we do not want to reason about fresh names not

belonging to the formula Φ, so we only need to be sure that, in every transition preceding the satisfaction of Φ, any fresh name is not free in the formula; the simplest way to do this is to shift free names of the formula by one position when a HD state performs a bound output transition; iterating this process is very similar to iterating the definition for the modalities of $\mathcal{F}$:

Definition 3.8 (Interpreting $E\,\Phi$ over HD-automata).

$$Q \vDash^{\rho} E\,\Phi \;\Leftrightarrow\; Q \vDash^{\rho} \Phi \vee \exists \sigma \in sym\,Q, \mu, \zeta, Q'.Q \overset{\mu}{\underset{\zeta}{\longmapsto}} Q' \wedge$$

$$Q' \vDash^{\zeta;(\sigma;\rho)+n} \Phi\, shift_n$$

where n is the number of fresh names generated along μ.

The definition for $shift_k$ is:

$$shift_k\,(x_i) = x_{i+k}$$

This process is decidable, as we will see in the chapter about model checking; as in the previous case, if $Q \vDash^{\rho} E\,\Phi$ then there exists a *finite* path $Q \overset{\mu_1}{\underset{\zeta_1}{\longmapsto}} \ldots \overset{\mu_n}{\underset{\zeta_n}{\longmapsto}} Q_n$ generating k fresh names such that, via a suitable name mapping, Q_n satisfies $\Phi\, shift_k$.

Adequacy, soundness and completeness

The introduction of the eventually modality does not affect adequacy, since if two agents, or two HD states, satisfy the same formulae in $\mathcal{F}^*$, then they also satisfy the same formulae in $\mathcal{F}$, and vice-versa.

Theorem 3.9 (Adequacy of $\mathcal{F}^*$). $\mathcal{F}^*$ is adequate for π-calculus early bisimulation and for HD-automata bisimulation.

Proof. We reason by induction over the structure of $\mathcal{F}^*$ formulae: if $f \notin \mathcal{F}^* \setminus \mathcal{F}$, certainly the property of adequacy holds, so we need only to show the result for a formula $E\,\Phi$, and can assume the induction hypothesis for Φ.

– Proof for π-calculus:

Suppose $P \sim P'$, and $P \vDash E\,\Phi$. Then by definition of $\vDash$ we know that a path $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} P_k$ of length $k \geq 0$ exists, such that $P_k \vDash \Phi$ and $fn\,(\Phi) \cap \left(\bigcup_{i\in[1,k]} bn\,(\alpha_i)\right) = \emptyset$. Now we can reason by induction over k; if $k = 0$ the property holds directly for induction hypothesis (the one over formulae, not the one over k).

$k > 0$

$\Rightarrow\ P \xrightarrow{\alpha_1} P_1$

$\Rightarrow\ \{P \sim P'\}$

$\exists \alpha_1', P_1'.P' \xrightarrow{\alpha_1'} P_1' \wedge P_1 \sim P_1'$

$\Rightarrow\ \{\text{induction hypothesis over } k, P_1 \vDash E\,\Phi\}$

$P_1' \vDash E\,\Phi$

$\Rightarrow\ P' \vDash E\,\Phi$

– Proof for HD-automata:

Suppose $Q \overset{\delta}{\sim} Q'$, and $Q \vDash^{\rho} E\,\Phi$. Again, there is a path $Q \overset{\alpha_1}{\underset{\zeta_1}{\longmapsto}} Q_1 \overset{\alpha_2}{\underset{\zeta_2}{\longmapsto}} \ldots \overset{\alpha_n}{\underset{\zeta_n}{\longmapsto}} Q_k$, and we reason by induction over its length k: If $k = 0$ the property holds for the main induction hypothesis, while for $k > 0$ we have:

$\exists \sigma \in sym(Q), Q_1, \mu, \zeta.Q \overset{\mu}{\underset{\zeta}{\longmapsto}} Q_1 \wedge Q_1 \vDash^{\zeta;(\sigma;\rho)_{+n}} E\,\Phi$

$\Leftrightarrow\ \left\{Q \overset{\delta}{\sim} Q'\right\}$

$\exists \sigma' \in sym(Q'), Q_1', \mu', \zeta'.Q \overset{\mu'}{\underset{\zeta'}{\longmapsto}} Q_1' \wedge Q_1 \overset{\left(\zeta;(\sigma;\delta;(\sigma')^{-1})_{+n};(\zeta')^{-1}\right)}{\sim} Q_1'$

$\Leftrightarrow\ \{\text{induction hypothesis over } k\}$

$Q_1' \vDash^{\left(\zeta';(\sigma';\delta^{-1};\sigma^{-1})_{+n};\zeta^{-1};\zeta;(\sigma;\rho)_{+n}\right)} E\,\Phi$

$\Leftrightarrow\ Q_1' \vDash^{\left(\zeta';(\sigma';\delta^{-1};\rho_{+n})\right)} E\,\Phi$

$\Leftrightarrow\ \{k > 0\}$

$Q' \vDash^{(\delta^{-1};\rho)} E\,\Phi$

□

Also our soundness and completeness results are unchanged, since if an agent P satisfies $E\,\Phi$ with k transitions, it satisfies a modal formula in $\mathcal{F}$ with k modalities, corresponding to the transitions in the path, and then Φ, by adequacy of $\mathcal{F}$. This formula is also satisfied by its corresponding HD state, via suitable name mappings.

3.6.2 Example: liveness and safety properties of a buffer

Now we go back to our running example, and exploit the new temporal operator to verify that the memory cell has a liveness property, in the stronger sense of "something good *always* happens": the cell never reaches a state in which it can't eventually read a value in the future. Also we check a safety property: it never happens that a certain value is read and immediately after a different value is outputted.

$$\begin{aligned} alive &= \neg E(\neg E\,\langle cx\rangle\, true) \\ safe &= \neg E(\neg E\,\langle cx\rangle\,\langle \overline{c}y\rangle\, true) \end{aligned}$$

These properties are true, but to prove it we should reason inductively about any possible path. It turns out that model checking a path modality over an automata is decidable, so model checking can prove these formulae without having to build an inductive proof.

Chapter 4

Model checking $\mathcal{F}$ and $\mathcal{F}^*$

4.1 Decidability of model checking

We will now take in exam decidability of a modal formula of $\mathcal{F}^*$ over a finite HD-automaton. There are, to the aim of finite-state model checking, many possible flaws in our definitions. First, symmetries in HD-states are infinite, and in the interpretation of $\mathcal{F}$ modalities we have used an existential quantification over a symmetry. Thus, we have to show that this process is decidable, by exploiting the fact that we only deal with *finite support* symmetries. Another possible problem is in the definition for the "eventually" operator: in this case, we have to check a possibly infinite number of paths, bounded by their length k, but we will see a way to make this process decidable.

How can we describe in a finite way all the mappings by which a HD state can satisfy a given modal formula $\langle\alpha\rangle A \in \mathcal{F}$? With a little abuse of notation we write $supp(Q)$ in place of $supp(sym(Q))$, and we define the shorthands:

$$\begin{aligned} S(Q, \langle\alpha\rangle A) &\stackrel{\text{def}}{=} \left\{\rho \middle| Q \vDash^{\rho} \langle\alpha\rangle A\right\} \\ ssym(Q) &\stackrel{\text{def}}{=} \left\{\rho \in sym(Q) \middle| \forall n \notin supp(Q).\rho(n) = n\right\} \\ nsym(Q) &\stackrel{\text{def}}{=} \left\{\rho \middle| \forall n \in supp(Q).\rho(n) = n\right\} \end{aligned}$$

Equation 2.2 can be rewritten as:

$$supp(S) = N \Rightarrow S = \left\{\rho; \rho' \middle| \rho \in ssym(Q) \wedge \rho' \in nsym(Q)\right\}$$

The important thing to note is that names of the support are always mapped, by a generic $\rho \in sym(Q)$, into names of the support and vice-versa. There is no

possibility for names into and outside the support to get mixed by a permutation in the symmetry of ρ.

We have:

$$\rho \in S(Q, \langle\alpha\rangle\, A)$$

$$\Leftrightarrow\ \exists\sigma \in sym(Q).\exists Q', \mu, \zeta.Q \overset{\mu}{\underset{\zeta}{\mapsto}} Q' \wedge (\sigma;\rho)(\mu) = l(\alpha) \wedge Q' \vDash^{\zeta;(\sigma;\rho)_{+n}} A\,rot_n$$

$$\Leftrightarrow\ \exists\sigma \in sym(Q).\exists Q', \mu, \zeta.Q \overset{\mu}{\underset{\zeta}{\mapsto}} Q' \wedge (\sigma;\rho)(\mu) = l(\alpha) \wedge (\zeta;(\sigma;\rho)_{+n}) \in S(Q', A\,rot_n)$$

$$\Leftrightarrow\ \exists\sigma_1 \in ssym(Q).\exists\sigma_2 \in nsym(Q).\exists Q', \mu, \zeta.Q \overset{\mu}{\underset{\zeta}{\mapsto}} Q' \wedge (\sigma_1;\sigma_2)(\mu) = \rho^{-1}(l(\alpha))$$

$$\wedge(\zeta;(\sigma_1;\sigma_2;\rho)_{+n}) \in S(Q', A\,rot_n)$$

Now we argue that the only part that matters for a permutation ρ to verify the assertion $Q \vDash^\rho A$ is the restriction of ρ to $fn\,(Q)$. This is true, indeed, because $\mathfrak{N} \setminus fn(Q)$ is outside the support of Q, so all the permutations of this set are in $sym(Q)$, and the fact that ρ maps one of those names into any other adds no further information. Moreover, the normalization mapping ζ sends $fn\,(Q')$ exactly to $fn(Q) \cup fn(\mu)$ shifted by n positions, plus the first n names.

Notice that σ_2 is important in the second half of the conjunction, according to the previous consideration, only for the binding between $fn\,(\mu) \setminus fn\,(Q)$ and $fn\,(\alpha)$, which is completely determined by the first half of the conjunction. Since the permutations mapping free names of Q into themselves are in a finite number, the possible interesting permutations $\sigma = \sigma_1;\sigma_2$ are also finite and obtained from $ssym(Q)$ by adding to each permutation in this set the correspondences obtained resolving the equation $\sigma(\mu) = \rho^{-1}(l(\alpha))$. This reasoning can be applied inductively to the second part of the conjunction to obtain a proof of decidability.

Regarding the "eventually" modality, we will show in the next section that we can find an equivalence relation over permutations by which the enumeration of $\{\rho | Q \vDash^\rho A\}$, where ρ is taken modulo the equivalence relation, is finite. This means that, when looking for a path originating in Q to satisfy the formula $E\,\Phi$, we have only a finite number of possibilities, and this makes model checking decidable.

4.2 The model checking algorithm

Model checking can be seen as the decisional problem of finding the paths in the state graph that satisfy a conjunction of formulae. The fact that there are

many *overlapping subproblems* leads to using *dynamic programming* to solve this problem: solutions to subproblems are not calculated each time they are required, but rather stored in a table, or cached applying *memoization* if not all possible subproblems are required to solve the main problem.

In our case, every time we need to check if state S of a given HD-automata satisfies formula A via mapping ρ, we have to build a three dimensional table $T[s, \sigma, a]$, where s is a HD state, a is a subformula of A and σ is a permutation. To reduce the required checks and optimize the algorithm, we can always use normalized formulae, using lemma A.3:

$$\forall A, Q, \rho. norm(A) = \langle A', \sigma \rangle \Rightarrow Q \vDash^{\rho} A \Leftrightarrow Q \vDash^{(\rho;\sigma^{-1})} A'$$

Since permutations which can map names of Q into names of A are infinite, (for example in the case of $Q \vDash^{\rho} true$, where ρ can be *any* permutation) we cannot index the third dimension of the table with any permutation, but have to choose permutations modulo an equivalence relation. We build the equivalence class of permutations which allow a generic state Q to satisfy a formula A:

$$\rho \equiv_{\langle Q,A \rangle} \rho' \Leftrightarrow \rho\big|_{r(Q,\rho,A)} = \rho'\big|_{r(Q,\rho',A)}$$

where

$$r(Q, \rho, A) = fn\,(Q) \cap \rho^{-1}(fn\,(A))$$

The following observation is the key to find an upper bound to the size of the table:

$$\rho \equiv_{\langle Q,A \rangle} \rho' \Rightarrow \forall Q. \forall A. Q \vDash^{\rho} A \Leftrightarrow Q \vDash^{\rho'} A$$

Since it does not matter how names outside $fn\,(Q) \cap \rho^{-1}(fn\,(A))$ are mapped, we can represent all the possible interesting permutations in a row of our table in a finite way by enumerating all the *partial* functions from $fn\,(Q) \cap \rho^{-1}(fn\,(A))$, which are in a finite number, even if it's exponential w.r.t. the maximum number of free names in a formula or in a state. In the implementation, this step will be done using a normalization function that, given a permutation ρ, a state Q and a formula A, returns a canonical representative of the class of equivalence of ρ over Q and A.

Having a finite table means that we do not have to check an infinite number of states when, starting from a state Q, we explore the graph of all triples of states, formulae and permutations reachable from $Q \vDash^{\rho} E\,\Phi$ in search for a state which

satisfies Φ via the right name mapping: if we discover that we have visited all of the reachable cells of the table, then we have found that $Q \not\approx^{\rho} A$.

Of course, the table will be very sparse over the dimension indexed by permutations, and a more appropriate data structure, which should be chosen depending on how one represents a finite support symmetry, could be used, however this does not affect the presentation of the algorithm, which we give below using the Caml programming language. Self-explaining auxiliary functions will be used without definition, but we will see their types after the main algorithm. Being this a reference implementation, we will not care too much of optimizations (e.g. redundant calls to the normalization function).

First of all, we introduce types used throughout the implementation:

```
type name
type subst
type perm
type state
type action
type formula = And of formula list
             | Not of formula
             | Next of action * formula
             | Eventually of formula
type label
type table
```

The only interesting one is the abstract syntax for formulae, while the other types are considered abstract.

The entry point of the program takes a state, a permutation and a formula, and returns the boolean value corresponding to whether the state satisfies or not the formula via the given mapping:

```
let rec model_checker : state -> perm -> formula -> bool =
 fun state rho formula ->
  let table = build_table state formula in
    memo table state rho formula
```

Initially the caching table is built (the function `build_table` will internally perform a normalization of the formula), and then the function `memo` is called. This function is responsible for normalizing the formula and the permutation and eventually calling the actual model checking routine, caching the results:

```
and memo : table -> state -> perm -> formula -> bool =
 fun table state rho formula ->
  let (fmla,p) = norm_f formula in
  let rho1 = norm_p (seq [rho;inv p]) state fmla in
   match (read_table table (state,rho1,fmla)) with
```

```
    Some x -> x
  | None ->
     let retval = check table state rho1 fmla in
       write_table table (state,rho1,fmla) retval;
       retval
```

The function, like all the following, takes the table as its first argument. The table has a function to read from it, which returns an optional boolean value representing whether the specified location has already been written or not, and the contents of the location in the first case. If no value is already present at the current coordinates, the formula and the permutation are normalized (the function `inv` is inversion of permutations). We assume that any uninteresting name is mapped by the resulting permutation to a special, unused value (which is one of the ways to represent partial functions). Then the model checking function is called, and the return value is stored in the table for later use. The model checking function performs a pattern matching operation on the formula and recursively calls the memoizing function:

```
and check : table -> state -> perm -> formula -> bool =
 fun table state rho formula ->
  match formula with
    And subfs ->
     for_all (memo table state rho) subfs
  | Not subf ->
     not (memo table state rho subf)
  | Next (act,subf) ->
     exists (check_next table state rho subf act)
       (out_transitions state)
  | Eventually subf ->
     if visited table (state,rho,formula)
      then false
      else if memo table state rho subf
            then true
            else
             begin
              set_visited table (state,rho,formula);
              exists
                (check_eventually table state rho formula)
                (out_transitions state)
             end
```

The `And` case is the one which allows the program to terminate when an empty conjunction (which, as we said, represents the boolean value *true*) is found. The `for_all` and `exists` functions are in the standard library of the language and check respectively for all or one element of a list to satisfy a boolean predicate.

The interesting cases are `Next` and `Eventually`. We find all the outgoing transitions from the current state, and check if any of these transitions satisfies the `check_next` or `check_eventually` function. The former will unify with a permutation the label of the transition and the action of the modal formula (applying ρ as in the definition), and cause recursion as expected, while the latter will be simpler, and only make the recursive call with appropriate mappings. As we already said, the eventually modality, according to the definition, should lead to an infinite computation but, when considering the graph given by all the cells of the table, and all the steps that we can make over those, it's clear that a cycle doesn't add information to the checking process, so we can perform a visit (a depth-first search, in this case) of this graph, using a flag in the table itself, which is initially clear, and can be set to indicate that the given cell has already been visited. Note that there are no problems with a formula like $E\,\Phi \wedge E\,\Phi$, where Φ is true in the starting state, because in the second call the memoization function returns true without allowing the program to check the *"visited"* flag again (which would erroneously lead to return *false* to the caller because the cell has already been visited).

We first see `check_next`:

```
and check_next : table -> state -> perm -> formula ->
                   action -> (state * label * perm) -> bool =
 fun table state rho subf act (dest,label,z) ->
  match make_equal label (apply_l (inv rho) (l act)) with
    None -> false
  | Some pairs ->
     let sigmas = find_compatible_perms state pairs in
     let fn = fresh_names act in
     let n = length fn in
     let fmla = apply_f (rot fn) subf in
       exists
         (fun sigma -> memo table dest
            (seq [z;plus n sigma;plus n rho]) fmla)
         sigmas
```

The first call is to the function `make_equal`, which takes two labels l_1 and l_2 and returns an optional list of pairs, explicitly enumerating the necessary bindings to turn l_1 into l_2. If the value `None` is returned, the arguments are not unifiable. Notice that this procedure is not proper *unification* [11], because the returned substitution must be injective. We look for a substitution σ such that $\texttt{label} = \sigma^{-1}(\texttt{rho}^{-1}(l(\texttt{act})))$, which is the same as saying $\sigma(\texttt{label}) = \texttt{rho}^{-1}(l(\texttt{act}))$. If no such substitution is found, we return false, else we find, in the symmetry of the current state, all the permutations `sigmas` that satisfy the substitution (which can be seen as a set of constraints). Then names of the immediate subformula that we are considering are rotated as in definition 3.3, and finally we check for

the existence of a permutation, among the `sigmas`, which satisfies the recursive invocation of the function `memo`, with the correct permutation as an argument (remember that normalization is performed in the `memo` function itself).

The implementation of `check_eventually` follows. The function `class_of` gives the number of fresh names for a HD label, i.e. the index i such that $\mu \in \mathfrak{L}_i$

```
and check_eventually : table -> state -> perm -> formula ->
                       (state * label * perm) -> bool =
 fun table state rho subf (dest,label,z) ->
     let n = class_of label in
     let fmla = apply_f (shift n) subf in
       exists
         (fun sigma -> memo table dest
            (seq [z;plus n sigma;plus n rho]) fmla)
         (find_useful_perms state
               (map_and_discard
                  (fun x -> applysubst (shift ~-n)
                         (applyperm z x)) (fn_s state))
               (map (applyperm (inv rho)) (fn_f fmla)) )
```

The problem when implementing the eventually modality is that, when we choose a permutation $\sigma \in sym(Q)$ (where Q is the current state), we don't add constraints, as instead we do in `check_next`, because it's not important where names of the transition label are sent. This way we should consider all the infinite permutations outside the support of $sym(Q)$. Consider for example the following π-calculus agent:

$$x(y).\overline{y}y.nil$$

This agent satisfies the formula $E\,\langle\overline{z}z\rangle\,true$, using permutation $\{x/y, y/x\}$ in the starting state; to avoid considering all the possible permutations, we select only those that map free names of the destination state of the transition we are considering into free names of the formula. Since there may be fresh names in the destination state, it's necessary to shift "backwards" those names, eventually loosing some name. The function `find_useful_perms` takes as arguments a state Q, a list of source names and a list of destination names, and returns all the permutations in $sym(Q)$ that are interesting for the given source and destination names (by an obvious enumeration). Names in the destination state of the transition are transformed through `z` and then shifted negatively, using the substitution `shift ~-n` applied by the function `applysubst`, which returns an option because the result could be undefined. The `map_and_discard` function takes f and l and applies f, which returns an optional value, to every element of l, returning all those elements which are not `None` (this function is not in the standard library

but it's easy to implement). Names of the formula are transformed through the inverse of `rho` as it's needed according to definition 3.3.

The main algorithm presentation could be ended here, however we also see the function `find_compatible_perms`, which deserves interest because it is the only point where we solve path constraints in the algorithm. It can be described as follows:

```
and find_compatible_perms : state -> (name * name) list ->
                                perm list =
  fun state pairs ->
    if exists
        (fun (a,b) ->
           mem a (support state) <> mem b (support state))
        pairs
    then []
    else
      let (filt,ext) = partition
                       (fun (a,b) ->
                             mem a (support state) &&
                             mem b (support state))
                       pairs in
      let to_extend = fold_left
                       (fun perms binding ->
                         (filter (perm_has binding) perms))
                       (ssym state) filt in
      let addition = build_permutation ext in
        map (fun p -> seq [p;addition]) to_extend
```

The function is based on the idea that, given an HD state Q, if we want to select all the permutations $\sigma \in sym(Q)$ such that $\sigma(a) = b$, then there are three possibilities:

1. One of a and b is in the support of Q, and the other is not. In this case, no permutation in $sym(Q)$ can satisfy the constraint.

2. Both a and b are in the support of Q. In this case we have to select from $ssym(Q)$ all those permutations σ which satisfy $\sigma(a) = b$, and no condition is imposed for names outside the support, so that we still have a finite support symmetry but with less permutations over the support.

3. None of a and b is in the support of Q. In this case, there are infinite permutations in $nsym(Q)$ that satisfy the constraint, so we choose the one that satisfies *all* the constraints of this kind simultaneously and is the identity on all other names (it does not matter, anyway) and compose this permutation with all the permutations obtained from the previous step.

The implementation first checks if the set of constraints is satisfiable. If it is, the list of pairs is partitioned into `filt`, which is the list of pairs in the support, and `ext`, the list of pairs outside the support. The formers are used to filter the permutations over the support of the current state, while the remaining bindings are used to build a single permutation, which is composed with the results of the previous step.

The function `mem` checks membership in a list, `fold_left` is a generic iterator over lists using a local state for accumulation of the results, `partition` splits a list into a pair of lists according to a boolean predicate, and the function `map` applies a function to all the elements of a list, returning the list of results. All these functions are in the standard Caml library.

A final consideration is about the handling of the caching table: should an implementor wish to use a more efficient representation, for all the symmetries in a row of the table, than explicit enumeration, this could be achieved in the implementation of the functions to build, read and write a table, without affecting the rest of the algorithm.

Types of auxiliary functions are as follows:

```
(* Functions to build and use tables *)
val build_table : state -> formula -> table
val read_table : table -> (state * perm * formula)
                  -> bool option
val write_table : table -> (state * perm * formula)
                   -> bool -> unit
val visited : table -> (state * perm * formula) -> bool
val set_visited : table -> (state * perm * formula) ->
                   unit

(* Functions over states and formulae *)
val out_transitions :
  state -> (state * label * perm) list
val norm_f : formula -> (formula * perm)
val support : state -> name list
val ssym : state -> perm list
val find_useful_perms : state -> name list -> name list ->
                          perm list
val fn_s : state -> name list
val fn_f : formula -> name list

(* Functions over substitutions *)
val apply_f : subst -> formula -> formula
val apply_l : perm -> label -> label
val applyperm : perm -> name -> name
val applysubst : subst -> name -> name option
```

```
val inv : perm -> perm
val rot : name list -> subst
val shift : int -> subst
val plus : int -> perm -> perm
val seq : perm list -> perm
val norm_p : perm -> state -> formula -> perm
val perm_has : (name * name) -> perm -> bool
val build_permutation : (name * name) list -> perm

(* Functions over actions and labels *)
val l : action -> label
val make_equal : label -> label -> (name * name) list option
val fresh_names : action -> name list
val class_of : label -> int

(* Other functions *)
val map_and_discard : ('a -> 'b option) -> 'a list -> 'b list
```

4.3 Complexity of the algorithm

As we said earlier, explicit representation of all the possible permutations gives a clear view of the algorithm, but it's definitely not good from the point of view of complexity. The number of partial bijective (if restricted to the right domain) functions from a set N of n elements to itself is obtained by choosing all the possible pairs of subsets of N of the same cardinality, and for any of those choosing all the possible permutations; the exact number is

$$\mathcal{P}_f(n) = \sum_{k=0}^{n} \binom{n}{k} \binom{n}{k} k!$$

which surely has a lower bound in $2^n = \sum_{k=0}^{n} \binom{n}{k}$. Of course, there is no necessity to explicitly store in memory all the possible permutations: we can use, as the third dimension of our table, a dynamic data structure (a bit like saying that a function from strings of length k into natural numbers can be represented explicitly as a table containing all the possible strings, or as a hash table). If we still have to examine all the possible permutations, this optimization does not change complexity.

We observe that, if we exclude the "eventually" modality which is the more problematic, the set of symmetries taken in exam at each step does not grow, so complexity of the algorithm on that dimension is bound by a more tractable number, the maximum cardinality of $ssym(Q)$ (which is often very small). Perhaps the exponential worst-case complexity is intrinsic to the problem, because

symmetries are a compact way to represent a series of states using a single one, but in model checking we enumerate all of the *system* states, so it may well be that this "unfolding" is inevitable; further study on the topic is worth.

Said this, to evaluate the worst-case complexity of our algorithm we can imagine to fill the whole table, starting from the bottom, since memoization is only an optimization. Be Q the set of states reachable from the starting one, q its cardinality, q' the maximum number of outgoing transitions starting in any state of Q, F the set of subformulae of the checked formula, f its cardinality, f' the maximum number of *immediate* subformulae of a formula in F, s the maximum cardinality of $ssym(S)$ for some S reachable from the starting state, n the maximum number of names in an element of $Q \cup F$, $p = \mathcal{P}_f(n)$.

The actual model checking function `check` is called no more times than the number of elements of the table, which is $q \cdot f \cdot p$, and to build the current state, when evaluating a formula of $\mathcal{F}$, we look at $q' \cdot f' \cdot s$ other cells. So total complexity is

$$(q \cdot f \cdot p) \cdot (q' \cdot f' \cdot s)$$

In the average case, $q' \cdot f' \cdot s$ should be much smaller than the remaining part (in particular, f' is 1 unless in case of a conjunction), so that we can consider it as a constant, and say that complexity is $q \cdot f \cdot p$.

Chapter 5

Case studies

We will take into exam some example, to test the expressivity of our temporal logic, and show that there are useful properties of a concurrent system with mobility of agents that can be modeled as $\mathcal{F}$ or $\mathcal{F}^*$ formulae. Since all the example programs are π-calculus agents, we have been able to take advantage from the previous work on HAL [26], and use its model checker to test our formulae.

Even if case studies presented here are not complicated (but the last one, which in the incorrect version has about two thousands of states), it would be impossible to hand check these requirements, while a machine can perform the checks in seconds, and even output a counterexample.

5.1 The dining philosophers

We will check the "no deadlock" and "no starvation" properties for the famous problem proposed by E. W. Dijkstra. Five philosophers are sitting at a round table, with a bowl of rice each one, and between each pair of philosophers is a chopstick. Philosophers either think or eat. To eat, a philosopher has to take two chopsticks, the one on his left and the one on his right. Then he eats, releases the chopsticks, and he gets back to thinking again, restarting the cycle.

For simplicity we will only check properties of a more intimate dinner with two philosophers; a stick is modeled by a process with two states: it first waits for an input, and then outputs a value on the channel received as input. Since our semantics is synchronous, writing a value on a channel means waiting for another process to read it, and this is what a philosopher process does to release the stick. A philosopher is modeled as a process with two channels, *eat* and *think*. The philosopher first writes a value (it does not matter what one) on *think*, then he acquires the two sticks in no particular order, he writes a value on *eat* to indicate he's eating, finally releasing the two sticks and going back to think.

$$
\begin{aligned}
Stick(x) &\stackrel{\text{def}}{=} x(y).\overline{y}y.Stick(x) \\
Ph(l,r,think,eat) &\stackrel{\text{def}}{=} \overline{think}\,think.(\bar{l}l.\bar{r}r.\overline{eat}\,eat.l(x).r(y).Ph(l,r,think,eat) \\
&\qquad + \bar{r}r.\bar{l}l.\overline{eat}\,eat.l(x).r(y).Ph(l,r,think,eat)) \\
Dinner(e_1,e_2,t) &\stackrel{\text{def}}{=} (\boldsymbol{\nu}r)\ (\boldsymbol{\nu}l)\ (Stick(l)|Stick(r)|Ph(l,r,t,e_1)|Ph(l,r,t,e_2))
\end{aligned}
$$

We have used a single channel as the *think* channel for both the philosophers, because we only will need to check if *someone* is thinking.

Now we can ask for the "no deadlock" property:

$$NoDeadlock = \neg E(\neg E(\langle \overline{t}t \rangle\, true))$$

The formula states that there will always, and infinite times, be someone who is writing over the channel "think" the value "think". This is expressed by stating the equivalent property that there will not eventually be a moment in which there is not eventually the possibility that someone write the value "think" on the channel "think".

The model checker returns *false* when verifying this formula. From literature, we already know that one possible solution to this problem is to impose an ordering over sticks:

$$Ph(l,r,think,eat) \stackrel{\text{def}}{=} \overline{think}\,think.\bar{l}l.\bar{r}r.\overline{eat}\,eat.l(x).r(y).Ph(l,r,think,eat)$$

In this case, our property holds. The other interesting property for the problem of the dining philosophers is the one known as "no starvation": we require (in the corrected version) that each man can eat for infinite times:

$$NoStarvation = \neg E\neg(E(\langle \overline{e_1}e_1 \rangle\, true) \wedge E(\langle \overline{e_2}e_2 \rangle\, true))$$

This formula, which is decidably true, is read as "there is no possible future situation in which it does not hold that both the events can eventually happen", where the events are $\overline{e_1}e_1$ and $\overline{e_2}e_2$.

5.2 Non atomic access to memory regions

We show a typical problem with shared memory: if two processes write simultaneously, into a random access memory, data which is constituted by more than one word, it may happen that data written by the two process is mixed. We show a didactic example where data is made up of two values, and each process can access two memory cells. The memory cell is modeled similarly to our buffer, but it has two different channels for input and output:

$$
\begin{array}{lcl}
Buf(i,o) & \stackrel{\text{def}}{=} & i(x).\overline{o}x.Buf(i,o) \\
Write(c_1,c_2,v_1,v_2) & \stackrel{\text{def}}{=} & \overline{c_1}v_1.\overline{c_2}v_2.\mathbf{0} \\
P(x,v_1,v_2,v_3,v_4) & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} i_1)\,(\boldsymbol{\nu} i_2)\;(\boldsymbol{\nu} o_1)\;(\boldsymbol{\nu} o_2) \\
 & & \qquad (Write(i_1,i_2,v_1,v_2)|Write(i_1,i_2,v_3,v_4)| \\
 & & \qquad\; Buf(i_1,o_1)|Buf(i_2,o_2)|o_1(a).o_2(b).\overline{x}a\overline{x}b)
\end{array}
$$

The fifth process, which writes on channel x, serves us as a tester for a safety property:

$$Integrity \quad = \quad \neg E(\langle \overline{x}v_1 \rangle \neg \langle \overline{x}v_2 \rangle true)$$

The model checker tells that this property is false. Then we might suspect what happens, and ask a more precise question:

$$Integrity_2 \quad = \quad \neg E(\langle \overline{x}v_1 \rangle \langle \overline{x}v_4 \rangle true)$$

This property is false in the starting state.

5.3 Mutual exclusion

Now we correct our implementation, by adding a *mutex*, i.e. a data structure which can be atomically locked and unlocked. We model a mutex as a process which reads a value from a fixed channel, then a value from the received channel, and then returns in its initial state:

$$Mutex(c) \quad \stackrel{\text{def}}{=} \quad c(x).x(y).Mutex(c)$$

A process uses the mutex by first generating a new name x, then sending it over c. Now the mutex is locked; to unlock it, the process, which is the only one entitled to do so, sends a value over x.

Here's our previous example, modified to lock memory before writing data. The $Write$ program has another argument, corresponding to the channel the mutex is listening on.

$$
\begin{array}{lcl}
Buf(i,o) & \stackrel{\text{def}}{=} & i(x).\overline{o}x.Buf(i,o) \\
Write(c_1,c_2,v_1,v_2,m) & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}x)\ \overline{m}x.\overline{c_1}v_1.\overline{c_2}v_2.\overline{x}x.\mathbf{0} \\
P(x,v_1,v_2,v_3,v_4) & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}i_1)\,(\boldsymbol{\nu}i_2)\ (\boldsymbol{\nu}o_1)\ (\boldsymbol{\nu}o_2)\ (\boldsymbol{\nu}m) \\
 & & \quad (Write(i_1,i_2,v_1,v_2,m)|Write(i_1,i_2,v_3,v_4,m) \\
 & & \quad Buf(i_1,o_1)|Buf(i_2,o_2)|o_1(a).o_2(b).\overline{x}a\overline{x}b| \\
 & & \quad Mutex(m))
\end{array}
$$

Now our two integrity properties are verified, and we can still check that something good eventually happens:

$$
SomethingGood \quad = \quad (E\,\langle\overline{x}v_1\rangle\,\langle\overline{x}v_2\rangle\,true) \vee (E\,\langle\overline{x}v_3\rangle\,\langle\overline{x}v_4\rangle\,true)
$$

5.4 An example involving mobility

We define a very simple mobile protocol, similar to the handover protocol described in [14], where a single client can access a system through one of three access points, and there is an input channel where anybody can send messages to this client.

The first program we introduce is the tracker, which keeps track of the channel where the client is reachable at. The tracker has two arguments, the *switch* channel, where access points register new locations for the client, and the *in* channel, where everybody can send messages to the client. The program has two states (the first one is used only before the client logs in for the first time), and it's similar to the memory cell of section 2.1.3:

$$
\begin{array}{lcl}
Tracker(switch,in) & \stackrel{\text{def}}{=} & switch\,(loc).Track2(loc,switch,in) \\
Track2(loc,switch,in) & \stackrel{\text{def}}{=} & in\,(msg).\overline{loc}\,msg.Track2(loc,switch,in)+ \\
 & & switch\,(newloc).Track2(newloc,switch,in)
\end{array}
$$

Now we introduce the access point, which knows the *switch* channel of the tracker, and has an *enter* channel, where the client can log in; the login procedure is very simple and does not take in account any security concern, such as the use of a password, because we will only check properties related to reachability of the client.

The login protocol, from the point of view of the client, consists in two steps:

1. The client sends a channel to the access point, to signal that he wants to log in.

2. On this channel, the client receives a new channel. This is the channel where he will receive new messages. From this point on, any previously received channel can be discarded and will never be used again.

From the point of view of the access point, before sending the new channel to the client the access point communicates it to the tracker via the *switch* channel:

$$Ap(switch, enter) \stackrel{\text{def}}{=} \quad enter\,(x).\,(\boldsymbol{\nu} c)\;\overline{switch}\,c.\overline{x}c.Ap(switch, enter)$$

The client knows the three *enter* channels of three access points, and at any time it can read a message and output it on a fixed channel (e.g. a message window on an user's screen), or log in to another access point:

$$\begin{aligned} Client(ap1, ap2, ap3, out) &\stackrel{\text{def}}{=} \\ &(c)(\overline{ap1}\,c.c(k).Comm(k, ap1, ap2, ap3, out) \\ &+\overline{ap2}\,c.c(k).Comm(k, ap1, ap2, ap3, out) \\ &+\overline{ap3}\,c.c(k).Comm(k, ap1, ap2, ap3, out)) \\ Comm(k, ap1, ap2, ap3, out) &\stackrel{\text{def}}{=} \\ &k(msg).\overline{out}\,msg.Comm(k, ap1, ap2, ap3, out) \\ &+Client(ap1, ap2, ap3, out) \end{aligned}$$

The main program puts in parallel composition a tracker in its starting state, three access points and a client. Again, we introduce a tester process, which writes the values x and y on the input port:

$$
\begin{aligned}
Prog2(out, x, y) \quad \stackrel{\text{def}}{=} \quad & (\boldsymbol{\nu} switch)\ (\boldsymbol{\nu} ap1)\ (\boldsymbol{\nu} ap2)\ (\boldsymbol{\nu} ap3)\ (\boldsymbol{\nu} in) \\
& (Ap(switch, ap1) | Ap(switch, ap2) | Ap(switch, ap3) \\
& | \, Tracker(switch, in) \\
& | \, Client(ap1, ap2, ap3, out) \\
& | \, \overline{in}\, x.\overline{in}\, y.\mathbf{0})
\end{aligned}
$$

Now we can finally test some property of the system; for first, we require that both x and y are eventually sent on the *out* channel, in the right order:

$$
\begin{aligned}
both \quad & = \quad E \left\langle \overline{out}\, x \right\rangle E \left\langle \overline{out}\, y \right\rangle true \\
ordered \quad & = \quad \neg E \left\langle \overline{out}\, y \right\rangle E \left\langle \overline{out}\, x \right\rangle true
\end{aligned}
$$

The model checker answers *true* to both questions. Then we check that no other value comes through the *out* channel, and of course the model checker agrees:

$$
correct \quad = \quad \neg E \left\langle \overline{out}\, z \right\rangle true
$$

Finally, we ask the model checker if it is possible that x is outputted and not y, i.e. if the channel can loose any message:

$$
lossless \quad = \quad \neg (E \left\langle \overline{out}\, x \right\rangle \neg E \left\langle \overline{out}\, y \right\rangle true)
$$

The model checker responds that this formula is true, so we can be happy with our mobile protocol. It is important to underline that this is a formal proof of correctness for the design of a relatively complex protocol, obtained by simple means that any programmer could easily learn and use. If the implementation is correct w.r.t. the design, the implementor can be mathematically sure that respecting the protocol no message will be lost; this is much better (when it can be done) than only testing the design against a set of test cases, but the specification language is very similar, so that already existing experience in testing can be easily reused with model checking.

5.5 Finding an error

Now we show that the model checker can really find errors in a protocol, by changing the program for the access point into:

$$Ap(switch, enter) \stackrel{\text{def}}{=} \quad enter\,(x).\,(\boldsymbol{\nu} c)\;\overline{x}c.\overline{switch}\,c.Ap(switch, enter)$$

We have inverted the two actions of communicating the fresh channel to the client and to the tracker. At first sight, this might seem not harmful because the channel is fresh. We check all of our properties and find that those all hold, but the last one: the channel may loose messages. Of course, what happens is that the client discards the old channel too early, before the tracker has stopped sending messages to it.

This example demonstrates how a possible problem, which, depending on the speed of the channel switch in the tracker process (specifically the nondeterministic choice in the $Track2$ program) might never show up in the testing phase, can be identified and corrected at design time without difficulty.

Besides, while testing and debugging concurrent programs after a test failure is often a complicated trial and error procedure, and as we said it's even possible that potential problems does not happen at all in the testing phase (when the program is run in a controlled environment, different from the one where the software will be used) model checking finds the problem and even returns a counterexample, so that an eventual mistake can be spotted and solved with ease.

Chapter 6

Conclusions and future work

Starting from the definition of a modal logic for the π-calculus, we have obtained a kind of interpretation for modal formulae over HD-automata which, under generalizable conditions (e.g. section 3.5), can be systematically adapted to many kinds of branching-time temporal logics. There are some features which are particularly suited to practical implementation of verification techniques using our approach:

- We have obtained a general framework to deal with finite-state enumerative methods for formal verification. The number of formalisms that can be mapped into the HD model, either directly, or by an intermediate encoding into π-calculus, leads to think that further studies in this area might provide an uniform theory for model checking algorithms.

- HD-automata with symmetries can be considered the implementative point of view of a recently introduced [19] coalgebraic semantics for the π-calculus, which used a systematic approach to obtain minimization. Minimizing a system before an exhaustive verification can become a crucial factor for the tractability of such problems.

- The absence of side conditions in the definition for modalities gives a uniform and intuitive treatment of modal formulae with name allocation, deallocation and passing. This is a direct consequence of the way HD-automata have been designed, and by itself is an evidence that that symmetries in states and locality of names are a good (if not minimal) abstract model for resource allocation and information passing mechanisms. The proof of adequacy itself, even if appearing mechanical, is in its most part a series of equations on substitutions; also in this case, the absence of side conditions helps in reasoning about the formalism.

The complexity result is, as we said, not so promising, but aligned with results using other methods, being the problem intrinsically enumerative. On the other hand, the logic is expressive: as case studies highlight, interesting properties of a system can be checked in reasonable time, and without the need to provide an hand made proof.

Being able to verify calculi with name passing and generation is indeed a very useful feature in software design. Since HD-automata have *local* names, only name identity is preserved in transitions, but names can represent many different things, such as objects, localities, channels or messages. Identity of objects, channel and messages is the key property to correctly represent a wide spectrum of programming paradigms at the design level, when one often does not want to deal with actual values of data fields but rather to their identity when operations are applied to them. With transmission of channels, we also achieve verification of programs involving mobility of code, which is, as we already said, an important challenge for computer science, given the recent broad diffusion of mobile devices of any kind.

There are many possibilities to extend the work presented here:

- Studying modal logics for other process calculi (e.g. with localities, causality etc.) and their translation into corresponding logics for HD-automata.
- In our formalism, names are local, and the only property they have is identity: if two names are distinct, they will never be mapped into equal names in the destination state of a transition. Ongoing research is working on modeling name *fusions* with HD-automata, and it would be interesting to see if the results of this thesis can be generalized to logics with fusion.
- Since we can map very different languages into HD-automata, it would be useful to see how we could integrate different parts of the design, written in different languages, into a single HD-automaton and verify properties of the obtained heterogeneous system.
- Adding other path modalities, or generalizing our logic using fixpoint operators. Even if this might seem the most appealing direction, our results for complexity of model checking, and maybe even for decidability, are to be restated and might not hold at all.
- Exploiting properties of substitutions to optimize the algorithm. After all, we handle the set of symmetries acting on the support of a state by explicit enumeration. An optimization step would be to find monotonic (w.r.t. composition) operators, related to the interpretation of our logic, that could then be applied to the generators of this group instead of all of its elements.

– Finding optimized data structures to represent a set of permutations with efficient *insert* and *search* (based on a set of constraints) operations. This would be aimed to implement the third dimension of the caching table, and can reduce average time and space complexity of the algorithm.

Given all the safety and fault-tolerance problems encountered by software produced with traditional design technologies when impacting the *global computing age* that has just begun, we hope that this work will contribute to the start of new methodologies, with formal specification and verification as strong foundations, which will make software written in the near future more reliable, and allow users to trust, and with few risks depend upon, services and products which are going to pervade our lives.

Appendix A

Auxiliary theorems and definitions

Lemma A.1 (Injective substitution and π-calculus transitions).

$$\rho \text{ injective } \Rightarrow (P \xrightarrow{\alpha} P' \Leftrightarrow P\rho \xrightarrow{apply(\rho,\alpha)} P'\rho)$$

where $apply(\rho, \alpha)$ is $\rho(\alpha)$ but where **bound** names $n_1, \ldots, n_k$ of the label have been converted to $\rho(n_1), \ldots, \rho(n_k)$, which is necessary because these names become free in P'.

Lemma A.2 (Injective substitution and $\vDash$).

$$\rho \text{ injective } \Rightarrow (P \vDash A \Leftrightarrow P\rho \vDash A\rho)$$

Lemma A.3 (Injective substitution and $\vDash_{HD}$).

$$\beta \text{ injective } \Rightarrow (Q \vDash^{\rho} A \Leftrightarrow Q \vDash^{(\rho;\beta)} A\beta)$$

Lemma A.4 (Composition of $rot_{\{n_1,\ldots,n_k\}}$ and ρ_{+k}).

$$\rho \text{ bijective } \Rightarrow rot_{\{n_1,\ldots,n_k\}}; \rho_{+k} = \rho; rot_{\{\rho(n_1),\ldots,\rho(n_k)\}}$$

Lemma A.5 (Composition of $rot_{\{n_1,\ldots,n_k\}}$ and $unrot_{(k,i)}$). Be $unrot_{(k,i)}$ the substitution

$$unrot_{(k,i)}(x_j) = \begin{cases} x_{i+j} & \text{if } j \in [0, k-1] \\ x_{j-k} & \text{if } j \in [k, i] \end{cases}$$

and $x_{i-1} = max(fn\,(X))$. Then

$$X\, rot_{n_1,\ldots,n_k}\, unrot_{(k,i)} = X\{x_i/n_1, \ldots, x_{i+k-1}/n_k\}$$

A corollary is that $\left(rot_{x_i}; unrot_{(1,i)}\right) = id$.

Definition A.6 (Modal formula corresponding to a HD-label).
Given a HD label μ we can build a modality α such that $Q \vDash^{id} \langle\alpha\rangle\, true \Leftrightarrow \exists Q', \zeta . Q \overset{\mu}{\underset{\zeta}{\longmapsto}} Q'$ with the following function:

$$\begin{aligned} m(tau) &= \tau \\ m(in(x,y)) &= xy \\ m(out(x,y)) &= \overline{x}y \\ m(bout(x)) &= \overline{x}(x_0) \end{aligned}$$

the proof is trivial and comes directly from the definition of $\vDash^{\rho}$; notice that the choice of x_0 is completely arbitrary: we could have chosen any name since this name will be discarded. We will also need a more precise function m^*, with specified fresh names in the case for bound output; to do so, we define m^* as a function from labels to modalities, but indexed over a name:

$$\begin{aligned} m^*_y(bout(x)) &= \overline{x}(y) \\ m^*_y(l) &= m(l) \text{ in other cases} \end{aligned}$$

If we wanted to generalize the results of this thesis to any set of labels, not only those of HD_π, we would have to index m^* over an arbitrary enumeration of names, and not only over a single name, to handle the polyadic case.

Bibliography

[1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997. `http://citeseer.ist.psu.edu/article/abadi97calculus.html`.

[2] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. `http://doi.acm.org/10.1145/5397.5399`.

[3] M. Dam. Model checking mobile processes. *Lecture Notes in Computer Science*, 715:22–36, 1993. `http://citeseer.ist.psu.edu/185001.html`.

[4] N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972.

[5] G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 2004. `http://matrix.iei.pi.cnr.it/FMT/WEBPAPER/TR-57-HAL.ps`. to appear.

[6] G. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, 1996.

[7] G. Ferrari, U. Montanari, and E. Tuosto. Coalgebraic minimisation of hd-automata for the π-calculus in a polymorphic λ-calculus. *Submitted for publication.* `http://www.it.uu.se/profundis/Year2/Deliv2/A.1.2.11.pdf`.

[8] S. Gnesi and G. Ristori. A model checking algorithm for π-calculus agents. In *Proc. Second International Conference on Temporal Logic (ICTL '97)*. Kluwer Academic Publishers, 1997.

[9] J.Y. Halpern and M.Y. Vardi. Model checking vs. theorem proving: a manifesto. pages 151–176, 1991. http://citeseer.ist.psu.edu/halpern91model.html.

[10] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996. http://citeseer.ist.psu.edu/lowe96breaking.html.

[11] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982. http://doi.acm.org/10.1145/357162.357169.

[12] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[13] R. Milner. Function as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[14] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993. http://citeseer.ist.psu.edu/milner91polyadic.html.

[15] R. Milner. What's in a name? (in honour of Roger Needham). 2003. http://www.cl.cam.ac.uk/~rm135/wosname.pdf.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i. *Information and Computation*, 100(1):1–40, 1992. http://www.lfcs.inf.ed.ac.uk/reports/89/ECS-LFCS-89-85/ECS-LFCS-89-85.p%s.

[17] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993. http://citeseer.ist.psu.edu/milner93modal.html.

[18] U. Montanari and M. Pistore. History-dependent automata. Technical Report TR-98-11, Dipartimento di Informatica, Pisa. ftp://ftp.di.unipi.it/pub/Papers/pistore/hdaut.ps.gz.

[19] U. Montanari and M. Pistore. Structured coalgebras and minimal hd-automata for the pi-calculus. *Theoretical Computer Science*, To appear. http://citeseer.ist.psu.edu/montanari00structured.html.

[20] U. Montanari and M. Pistore. Minimal transition systems for history-preserving bisimulation. In Proc. STACS'97*, LNCS 1200*. Springer Verlag, 1997.

[21] M. Pistore. *History Dependent Automata*. PhD thesis, Università di Pisa, Dipartimento di Informatica, 1999. `http://www.di.unipi.it/phd/tesi/tesi_1999/TD-5-99.ps.gz`. available at University of Pisa as PhD. Thesis TD-5/99.

[22] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[23] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. `http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/mypapers.html%`.

[24] D. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.

[25] R. Bartolini. Model-checking di proprietà causali di reti di petri. Master's thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[26] *web site*. *The HD-automata laboratory*. `http://fmt.isti.cnr.it:8080/hal`.

[27] *web site*. *The Mobility Workbench*. `http://www.it.uu.se/research/group/mobility/mwb`.