APPENDIX 1.3.3

G. Ferrari, U. Montanari, R. Raggi, and E. Tuosto. From coalgebraic specification to toolkit development. Technical Report TR-02-19, Technical Report, Dipartimento di Informatica Universita' di Pisa, 2002.

Università di Pisa
Dipartimento di Informatica

Technical Report: TR–02–19

From Coalgebraic Specification to Toolkit Development

Gianluigi Ferrari   Ugo Montanari   Roberto Raggi
Emilio Tuosto

December 19, 2002

ADDRESS: Via F. Buonarroti 2, 56127 Pisa, Italy.
TEL: +39 050 2212700 — FAX: +39 050 2212726

From Coalgebraic Specification to Toolkit Development

Gianluigi Ferrari Ugo Montanari Roberto Raggi
Emilio Tuosto

December 19, 2002

Abstract

This paper describes the architecture of a toolkit performing state minimization of labelled transition systems for name passing calculi. The structure of the toolkit is developed from the co-algebraic formulation of the partition-refinement minimization algorithm. Indeed, the concrete software architecture of the minimization toolkit is directly suggested by the abstract semantical structure of the coalgebraic specification. The direct correspondance between the semantical structures and the implementation structures facilitates the proof of correctness of the impementation. We evaluate the usefulness of the minimization toolkit in practice by performing finite state verification of pi-calculus specifications.

keywords: Formal Verification, Name Passin Process Calculi, Partition Refinement, Semantic-based Verification Environments

Contents

1 Introduction **3**

I Backgrounds **5**

2 The π-Calculus **6**
- 2.1 Syntax . . . 6
- 2.2 Early semantics of π-calculus . . . 8
- 2.3 Late semantics . . . 11
- 2.4 Variants of π-calculus . . . 12

3 Categories and Functors **13**

4 Algebras and coalgebras **17**

5 Transition Systems as Coalgebras **18**

6 A comparison **21**

7 HD-automata for π-agents **23**
7.1 Bundles over π-calculus actions 24
7.2 The minimization algorithm 27

II Mihda: A Verification Environment **29**

8 Architectural Aspects of Mihda **29**

9 Main data structures **31**
9.1 HD-automata states, labels and transitions 32
9.2 Block 37

10 The main cycle **40**

11 Concluding Remarks **44**

1 Introduction

Verification of systems that can be adequately modeled as mobile processes is difficult because many "source of infinity" can be introduced. For instance, let us consider transition systems obtained from π-agents, we have that transitions can generate new names. Indeed, let us consider the $(OPEN)$ rule of π-calculus:

$$\frac{p \xrightarrow{\overline{x}y} q}{(\nu\ y)p \xrightarrow{\overline{x}(y)} q} \quad \text{if } x \neq y. \tag{1}$$

Such rule basically establishes that a state $(\nu\ y)p$ can create a new name and can export it over a channel x. Notice that the state of the transition system corresponding to $(\nu\ y)p$ represents a point of the computation where y "does not exists", while the target state of the bound output transition is a point of the computation where y "becomes available". Rule (1) introduces an infinite branching in the automata corresponding to agents that perform bound output transitions.

The rule for input transition of the early semantics of the π-calculus also introduces infinite branching because it is necessary to consider a transition for *any* name that instantiates the input parameter.

Let us remark that it is of course reasonable (and desirable) to model π-calculus semantics with rules as (1) or by means of the early semantics because such rules account for scope extrusion of names that is one of the major peculiarities of π-calculus and permits to model and reason on many aspects of mobile systems. On the other hand, since those kind of semantics had been introduced without considering verification issues, such rules are problematic when verification purposes are under consideration.

A different phenomenon that produces infinite automata is due to name extrusion in relation with recursion. A possible "implementation" of name extrusion is to reserve an infinite sequence of names from which a new name can be taken when a transition extrudes a fresh name. This approach has been proposed and analyzed in [33, 12]. A drawback of this approach is that an infinite number of states is generated in the case of agents with infinite behaviour. Indeed, let us consider the agent $A(x) = (\nu\ y)\bar{x}y.A(y)$. Agent $A(x)$ generates a new name y, emit it along x and continues as $A(y)$. This behavior is "encoded" in the approach of [33, 12] as

$$A(x) \xrightarrow{\bar{x}(x_0)} A(x_0) \xrightarrow{\bar{x_0}(x_1)} A(x_1) \xrightarrow{\bar{x_1}(x_2)} A(x_2)...$$

Hence, to obtain finite state automata also for agents with infinite behaviour, we need a mechanism to model "resource deallocation". Let us again consider agent A; after each bound output $\bar{x}_i(x_{i+1})$ transition, the name x_i will never be used in future transitions, hence we could re-use it provided that a mechanism for re-stating its freshness is given.

In order to model this kind of evolution in a framework suitable for verifying systems it is necessary to enrich the structure of states and transitions of

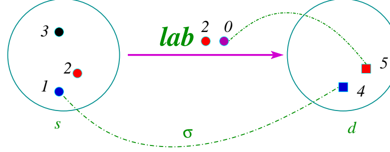


Figure 1: A HD-automaton transition

ordinary transition systems.

History Dependent automata (HD-automata in brief) have been proposed in [32, 27, 28, 11] as a new operational model for history dependent calculi, namely those calculi whose semantics is defined in terms of a labeled transition system such that the labels may carry information generated in the past transitions of the system and this "historical" information can influence the future behaviour of the system. Probably the simplest history dependent calculus is CCS with value passing [19], another example is the CCS with locality [6]; finally, as we have seen π-calculus LTS semantics all have labels that can contain names generated in past transitions[1].

HD-automata aim at giving a finite representation of otherwise infinite label transition systems. Similarly to ordinary automata, HD-automata are made out of states and labeled transitions. Their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt as syntactic components of labels, but become an explicit part of the operational model. This permits to model name creation/deallocation or name extrusion that are typical linguistic mechanisms of name passing calculi.

An important aspect of HD-automata to emphasize is that names of a state have *local meaning*. For instance, if $A(x, y, z)$ denotes an agent having three free names x, y and z, then agent $A(y, x, z)$ is different from $A(x, y, z)$, however, they can be represented by means of a single state q in a HD-automaton simply by considering a "swapping" operation on the names (corresponding to) x and y of q. More generally, states that differs only for renaming of their local names are identified in the operational model.

Local meaning of names requires a mechanism for describing how names correspond each other along transitions. Graphically, we can represent such correspondences using "wires" that connect names of label, source and target states of transitions. For instance, Figure 1 depicts a transition from source state s to destination state d. The transition exposes two names: Name 2 of s and a fresh name 0. State s has three names, 1, 2 and 3 while d has two names 4 and 5 which correspond to name 1 of s and to the new name 0, respectively. Notice that names 3 is discharged along such transition.

As described in Figure 1, HD-automata relies on the fact that names are local. This allows for a compact representation of agent behaviour by collapsing states that differ only for renaming of local names encompasses the main

[1] Also formalism that are not related to process calculi can be considered as history dependent; for instance, Petri nets [15] are a paradigmatic history dependent formalisms.

characteristics of name-passing calculi, namely, creation/deallocation of names. Indeed, name creation is simply handled by associating in the target state a name not in the source state.

A computation performed on a HD-automaton associates a "history" to names of the states appearing in the computation, in the sense that it is possible to reconstruct the associations which lead to the state containing the name. Clearly, if a state is reached in two different computations, different histories could be assigned to its names.

Various families of HD-automata have been introduced. Roughly speaking each class of HD-automata corresponds to a class of history dependent calculi or different behavioural semantics. The reader is referred to [32] for details.

In this paper we present Mihda, an implementation of a verification environment based on the co-algebraic formulation of the partition-refinement algorithm for HD-automata. The main result of the paper is to show how Mihda effectively is a refinement of the co-algebraic specification. We prove that the Mihda corresponds to the implementation of a functor mapping objects from the category of HD-automata to more concrete ocaml objects.

Another interesting result is the possibility of exploits Mihda as a "minimization" library, in the sense that Mihda exploits the *module system* of ocaml that gives the opportunity of applying the same algorithm to different kind of automata. For instance the minimization algorithm has been uniformly applied both to HD-automata and to ordinary automata (see [22]).

Outline of the paper. This paper is divided in two parts. The first parts collects some preliminary definitions and results related to π-calculus, category theory and co-algebras which aim at keeping this work self-contained (as much as possible). The second part introduces Mihda and discussed some implementation choices. We show how the co-algebraic framework can guide the implementation and helps maintaining strictly connected theoretical definitions and the corresponding concrete structures and functionalities.

Part I
Backgrounds

This part points out the theoretical frameworks in which we work. We first report some definitions and discussions on π-calculus discussing the most relevant aspects of its semantics with respect to verification issues. Then we detail elementary notions from category theory and co-algebras that will make more clear the presentation of our results. In particular, we point out how co-algebras can be suitably exploited for representing automata and for defining a semantic minimization algorithm. Finally, we report the results presented in [11] where HD-automata for π-agents and minimization algorithm for them have been introduced.

2 The π-Calculus

The π-calculus [25] is the best known example of core calculus for mobility. It is centered around the notion of *naming*: mobility is achieved via *name passing*. Channel names can be created, communicated and are subjected to sophisticated scoping rules. The capability of exchanging channel names gives π-calculus the ability of dynamically reconfiguring process acquaintances.

Name passing primitives are simple but expressive; indeed π-calculus can model objects (in the sense of object oriented programming [38]) and higher order communication [35].

In this section we outline the syntax and the early semantics of the calculus and refer the reader to [25, 36, 24] for a detailed presentation of the variegated facets of π-calculus.

2.1 Syntax

We assume as given an infinite set of names $\mathcal{N}$ and we let $a, b, \ldots, x, y, \ldots$ to range over $\mathcal{N}$. Agents of π-calculus are built over terms generated by the following productions:

$$\begin{array}{lcl} p & ::= & \mathbf{0} \ | \ \pi.p \ | \ p \,|\, q \ | \ p+q \ | \ (\nu\, y)p \ | \ [x=y]p \ | \ A(x_1, ..., x_n) \\ \pi & ::= & \tau \ | \ x(y) \ | \ \overline{x}y \end{array} \tag{2}$$

A process can be the void process, a process prefixed with actions, the parallel composition of processes, the non-deterministic alternative between two processes, a process obtained by restricting a name, a process guarded by equality of names or the recursive invocation of an agent. In (2) we let A to range over a set of process identifiers and, for each A, we assume that

- there is a unique definition $A(y_1, ..., y_n) \stackrel{\triangle}{=} q$ where the y_i's are all distinct and $\mathrm{fn}(q) \subseteq \{y_1, ..., y_n\}$;
- whenever A is used, its arity is respected;
- if $A(y_1, ..., y_n) \stackrel{\triangle}{=} p$ is the definition of A, each process identifier in p is in the scope of a prefix (guarded recursion).

Actions of π-calculus are

- τ, also called *silent* action, that represent non-observable or internal computation,
- *input action* $x(y)$ representing the reception along channel x of a name to be replaced for y,
- *output action* $\overline{x}y$ represents the output of name y along channel x.

For input and output action we call x the *subject* and y the *object* name, respectively.

π	$\text{fn}(\pi)$	$\text{bn}(\pi)$	$\text{n}(\pi)$
τ	$\emptyset$	$\emptyset$	$\emptyset$
$x(y)$	$\{x\}$	$\{y\}$	$\{x,y\}$
$\overline{x}y$	$\{x,y\}$	$\emptyset$	$\{x,y\}$

Table 1: Free and bound names of π-calculus prefixes

The input action and the restriction operator $x(y)._$ and $(\nu\, y)\ _$ act as binders for name y with scope the argument process. However, they have different nature: in the first case, y indicates the placeholders where the received name must be placed; in the second case, y is a new, private name. Notions of *free names* of a prefix action π, $\text{fn}(\pi)$, of *bound names* of π, $\text{bn}(\pi)$ arise as expected and are reported in Table 1. Given a process p, we can define *free names* of p, $\text{fn}(p)$ and *bound names* of π, $\text{bn}(\pi)$ as done in Table 2, where $A(y_1,...,y_n)\stackrel{\triangle}{=}q$. The set of *names* of p is the set $\text{n}(p) = \text{fn}(p) \cup \text{bn}(p)$. We shall write $\text{fn}(p,q)$ in

p	$\text{fn}(p)$	$\text{bn}(p)$
$\mathbf{0}$	$\emptyset$	$\emptyset$
$\pi.q$	$(\text{fn}(\pi) \cup \text{fn}(q)) \setminus \text{bn}(\pi)$	$\text{bn}(\pi) \cup \text{bn}(q)$
$q_1 \mid q_2$	$\text{fn}(q_1) \cup \text{fn}(q_2)$	$\text{bn}(q_1) \cup \text{bn}(q_2)$
$q_1 + q_2$	$\text{fn}(q_1) \cup \text{fn}(q_2)$	$\text{bn}(q_1) \cup \text{bn}(q_2)$
$(\nu\, y)q$	$\text{fn}(q) \setminus y$	$\text{bn}(q) \cup y$
$[x=y]q$	$\text{fn}(q) \cup \{x,y\}$	$\text{bn}(q)$
$A(x_1,...,x_n)$	$\text{fn}(x_1,\cdots,x_n)$	$\emptyset$

Table 2: Free and bound names of π-calculus processes

place of $\text{fn}(p) \cup \text{fn}(q)$ (similarly for $\text{bn}(\cdot)$ and $\text{n}(\cdot)$).

We adopt the following usual syntactic conventions: $\pi.p \mid q$ stands for $(\pi.p) \mid q$, $(\nu\, x)p \mid q$ for $((\nu\, x)p) \mid q$ and $(\nu\, x_1 \dots x_m)p$ for $(\nu\, x_1)\dots(\nu\, x_m)p$. Moreover, trailing occurrences of $\mathbf{0}$ shall usually be omitted.

A *structural congruence* relation, $\equiv$, is defined on π-calculus agents. It is the least congruence relation that satisfies the axioms in Table 3. The structural

(ALPHA)	processes which differ by α-conversion are equivalent
(PAR)	$\mid$ is associative and commutative, and $\mathbf{0}$ is its identity
(SUM)	$+$ is associative and commutative and $\mathbf{0}$ is its identity
(SCOPE)	$p \mid (\nu\, a)\ q \ \equiv\ (\nu\, a)(p \mid q) \quad$ if $a \notin \text{fn}(p)$
(RES)	$(\nu\, a)\ (\nu\, b)\ p \ \equiv\ (\nu\, b)\ (\nu\, a)\ p$
(NIL)	$(\nu\, a)\mathbf{0} \ \equiv\ \mathbf{0}$
(MATCH)	$[a=a]\mathbf{0} \ \equiv\ \mathbf{0}$

Table 3: π-calculus structural congruence

congruence basically provides an equational algebra for manipulating and rearranging processes without affecting their behaviour and simplifying the rules of operational semantics. For instance, process $a(x).\overline{x}b \mid (\nu\, y)\overline{a}y$ is structurally equivalent to $(\nu\, y)(a(x).\overline{x}b \mid \overline{a}y)$ because we can enlarge the scope of the ν operator. Moreover, structural congruence performs some garbage collection of dead processes.

It is worth to add some comments to the syntax of the calculus in order to give an intuition of the meaning of its terms before giving their formal semantics. Process $\mathbf{0}$ stands for a process that has no possibility of interacting with other processes. A process prefixed by an action $\pi.q$ can evolve after the action π has been fired; if π is the silent action, then the process evolves without synchronizing with other processes, otherwise another process must offer a complementary action[2] for the synchronization: Until such action is not offered, the process cannot evolve. Parallel composition of q_1 and q_2, $q_1 \mid q_2$, evolves to $q_1' \mid q_2$ when q_1 evolves to q_1' without interacting with q_2 (and similarly for q_2), or evolves to $q_1' \mid q_2'$ when q_1 and q_2 synchronize. Differently, if q_1 evolves to q_1' then non-deterministic choice $q_1 + q_2$ evolves to q_1' too disregarding the alternative q_2 (and similarly for q_2). As stated above, $(\nu\, y)p$ hides y to the processes outside the scope p; we will see that the scope of the restriction can dynamically change. Matching-guarded process $[x = y]p$ evolves as p only if the condition $x = y$ holds, otherwise it is stuck. Finally, if $A(y_1, ..., y_n) \stackrel{\triangle}{=} q$, then a recursive invocation $A(x_1, ..., x_n)$ is the same as $q[^{x_1,\cdots,x_n}/_{y_1,\cdots,y_n}]$, namely, as the process obtained by substituting each free occurrence of the formal parameter y_i with the actual parameter x_i.

2.2 Early semantics of π-calculus

The early semantics of π-calculus was first introduced in [26]. We report here a slightly simplified variation given in [32]. The labels of the labeled transition system for the early semantics of π-calculus are specified by the following productions:

$$\mu \quad ::= \quad \tau \quad \mid \quad xy \quad \mid \quad \overline{x}y \quad \mid \quad \overline{x}(y)$$

and are respectively called *synchronization*, *free input*, *free output* and *bound output* actions. Table 4 reports the definition of *free names*, *bound names* and *names* of a label μ, respectively written as $\mathrm{fn}(\mu)$, $\mathrm{bn}(\mu)$ and $\mathrm{n}(\mu)$. The labeled transition system for the early semantics of π-calculus is specified by the rules in Table 5. Let us remark that actions are different from prefixes because the free input and the bound output actions are not prefixes. Indeed input prefixes have the form $x(y)$ while free inputs are xy. The notation should be reminiscent of the fact that input prefixes act as binders for the object variables, instead the objects in free inputs are the effective received values in input actions. The bound output transitions are the peculiarity of the π-calculus. A bound output transition represent the communication of a name that has previously

[2] if π is an output on x the complementary actions are input actions on x, or else output actions on x, if π is an input on x.

μ	$\mathrm{fn}(\mu)$	$\mathrm{bn}(\mu)$	$\mathrm{n}(\mu)$
xy	$\{x,y\}$	$\emptyset$	$\{x,y\}$
$\overline{x}y$	$\{x,y\}$	$\emptyset$	$\{x,y\}$
$\overline{x}(y)$	$\{x\}$	$\{y\}$	$\{x,y\}$
τ	$\emptyset$	$\emptyset$	$\emptyset$

Table 4: Free and bound names of π-calculus labels

$$[\mathit{tau}] \quad \tau.p \xrightarrow{\tau} q \qquad\qquad [\mathit{out}] \quad \overline{x}y.p \xrightarrow{\overline{x}y} p$$

$$[\mathit{in}] \quad x(z).p \xrightarrow{xy} p[{}^{y}/{}_{z}] \qquad\qquad [\mathit{sum}] \quad \frac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'}$$

$$[\mathit{par}] \quad \frac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q} \text{ if } \mathrm{bn}(\mu) \cap \mathrm{fn}(q) = \emptyset \qquad\qquad [\mathit{comm}] \quad \frac{p \xrightarrow{xy} p' \qquad q \xrightarrow{\overline{x}y} q'}{p \mid q \tau p' \mid q'}$$

$$[\mathit{match}] \quad \frac{p \xrightarrow{\mu} p'}{[x=x]p \xrightarrow{\mu} p'} \text{ if } x \notin \mathrm{bn}(\mu) \qquad\qquad [\mathit{res}] \quad \frac{p \xrightarrow{\mu} p'}{(\nu\, x)p \xrightarrow{\mu} (\nu\, x)p'} \text{ if } x \notin \mathrm{n}(\mu)$$

$$[\mathit{open}] \quad \frac{p \xrightarrow{\overline{x}y} p'}{(\nu\, y)p \xrightarrow{\overline{x}(y)} p'} \text{ if } x \neq y \qquad\qquad [\mathit{close}] \quad \frac{p \xrightarrow{xy} p' \qquad q \xrightarrow{\overline{x}(y)} q'}{p \mid q \xrightarrow{\tau} (\nu\, y)(p' \mid q')} \text{ if } y \notin \mathrm{fn}(q)$$

$$[\mathit{rec}] \quad \frac{p[{}^{x_1,\cdots,x_n}/{}_{y_1,\cdots,y_n}] \xrightarrow{\mu} p'}{A(x_1,...,x_n) \xrightarrow{\mu} p'} \text{ if } A(y_1,...,y_n) \stackrel{\triangle}{=} p \qquad\qquad [\mathit{cong}] \quad \frac{p \equiv p' \qquad p' \xrightarrow{\mu} q' \qquad q' \equiv q}{p \xrightarrow{\mu} q}$$

Table 5: Early semantics of π-calculus

been restricted and, therefore, it corresponds to the generation of a name new with respect to "the names of the environment". This mechanism is called *name extrusion* and is formalized by the interplay between rule [*open*] and rule [*close*]. Rule [*open*] reads as: if p can perform a free output transition $\overline{x}y$ and continues as p' then $(\nu\, y)p$ can make a bound output transition $\overline{x}(y)$ and continues as p', provided that $x \neq y$. Note that after bound output transition y is no longer restricted. If a synchronization involving a bound output and a free input action takes place, after the transition we restrict again the "newly generated" name y. Side conditions of the rules [*par*], [*open*] and [*close*] are necessary for avoiding name capture of free names. The remaining rules, are basically the formalization of their informal description given at the end of Section 2.1.

Observation 2.1 *An important aspect concerning early semantics and verification of π-agents must be remarked. The peculiarity of early semantics lies in the rule [in] that instantiates the object name when the input transition is derived. This implies that a process $x(z).p$ can trigger an infinite number of transitions (one for each instantiated name y). If we think of π-agents as nodes in an automaton, this gives rise to infinite branch on any input node. We will discuss this issues in deeper detail in Chapter 1.*

Now we present the definition of early bisimulation for π-calculus.

Definition 2.1 (Early bisimulation) *A binary relation $\mathcal{R}$ over π-agents is an* early bisimulation *if, whenever $p\mathcal{R}q$ then*

> *for each $p \xrightarrow{\mu} p'$ such that* $\mathrm{bn}(\mu) \cap \mathrm{fn}(p,q) = \emptyset$, *there is some $q \xrightarrow{\mu} q'$ such that $p'\mathcal{R}q'$.*

Two π-agents are early bisimilar, *written $p \sim q$, whether there is a bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.*

Condition $\mathrm{bn}(\mu) \cap \mathrm{fn}(p,q) = \emptyset$ in Definition 2.1 is necessary to guarantee that the name chosen to represent the newly created name in a bound output transition is "fresh" for both agents. The following example should make more clear the need for name freshness in bound output transitions.

Example 2.1 *Let us consider the π-agents $p \equiv (\nu\, y)\overline{x}y.\mathbf{0}$. It is easy to see that the transition*

$$(\nu\, y)\overline{x}y.\mathbf{0} \xrightarrow{\overline{x}(y)} \mathbf{0}$$

can be inferred by rules [out] and [open] in Table 5. Intuitively, p should not be distinguished from

$$q \equiv (\nu\, y)\overline{x}y + (\nu\, z)\overline{z}w$$

Indeed, $q \xrightarrow{\overline{x}(y)} \mathbf{0}$ is the unique transition that can be inferred for q because, output on z is prevented by the restriction that violates side conditions of rules [res] and [open] and therefore, $\{(p,q),(\mathbf{0},\mathbf{0})\}$ is a bisimulation relation according to Definition 2.1. However, if we discard condition $\mathrm{bn}(\mu) \cap \mathrm{fn}(p,q) = \emptyset$ *we could chose w for the newly generated name of p that does not fit for q because w is not new for it. This would prevent q to match the transition $p \xrightarrow{\overline{x}(w)} \mathbf{0}$.*

$$[in^l] \quad x(y).p \xrightarrow{x(y)} p \qquad\qquad [comm^l] \quad \frac{p \xrightarrow{x(y)} p' \qquad q \xrightarrow{\overline{x}z} q'}{p \mid q \xrightarrow{\mu} p'[^z/_y] \mid q'}$$

$$[close^l] \quad \frac{p \xrightarrow{x(y)} p' \qquad q \xrightarrow{\overline{x}(y)} q'}{p \mid q \xrightarrow{\tau} (\nu\, y)(p' \mid q')}$$

Table 6: Late semantics of π-calculus

2.3 Late semantics

Late π-calculus, is an alternative semantics given in [25]. The main difference between late and early semantics is "the moment" at which input names are instantiated. Rule $[in]$ in Table 5 states that y is substituted for z when the input prefix is encountered. On the contrary, the late semantics instantiates it only when a synchronization effectively takes place.

We outline the late semantics of π-calculus "by difference" with respect to the early semantics reported in Section 2.2. The *late actions* that an agent can perform are defined as:

$$\mu ::= \tau \quad | \quad x(y) \quad | \quad \bar{x}y \quad | \quad \bar{x}(y).$$

The only difference with respect to the labels of the early semantics is the *bound input* action. The parenthesis should remind that y does not represent a name; it will be used as a "placeholder" that indicates where the effectively received name should be substituted in the continuation. Hence, we have that $\mathrm{fn}(x(y)) = \{x\}$ and $\mathrm{bn}(x(y)) = \{y\}$.

The transition rules remain the same of the early semantics apart from those reported in Table 6. The reader can notice that rule $[in^l]$ simply declares that an input action can be performed by a process without instantiating the formal parameter y to any actual value. *Late*r, when a free output action (rule $[comm^l]$) will synchronize on name x, the actual name z will be substituted for y in the continuation of the input process. Rule $[close^l]$ is similar but it does not requires any instantiation because bound input and output transitions can always be renamed such that the bound names are turned into the same name.

An interesting exercise is to compare the natural bisimulation relation that arises from the late semantics.

Definition 2.2 (Late bisimulation) *A binary relation $\mathcal{R}$ over π-agents is a* late bisimulation *if, whenever $p\mathcal{R}q$ then*

- *for each $p \xrightarrow{\mu} p'$ with $\mu \neq x(y)$ and $\mathrm{bn}(\mu) \cap \mathrm{fn}(p, q) = \emptyset$, there is some $q \xrightarrow{\mu} q'$ such that $p'\mathcal{R}q'$;*
- *for each $p \xrightarrow{x(y)} p'$ with $y \notin \mathrm{fn}(p, q)$ there is some $q \xrightarrow{x(y)} q'$ such that, for all $z \in \mathcal{N}$, $p'[^z/_y]\mathcal{R}q'[^z/_y]$*

Two π-agents are late bisimilar *whether there is a late bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.*

The first thing to remark is that for non-input transitions, late and early bisimulation are defined in the same manner, namely, the first clause of Definition 2.2 is the same of the one in Definition 2.1. For input actions definitions differ each other; indeed, late bisimulation is stronger that early bisimulation because if $p \xrightarrow{x(y)} p'$ the choice of a transition $q \xrightarrow{x(y)} q'$ must not depend on the received name z. On the contrary, for the early semantics, we choose a free input transition of q depending on the received name in the transition of p.

Example 2.2 *Let us consider the following π-agents:*

$$q = x(y).\tau.\mathbf{0} + x(y).\mathbf{0} \qquad \textit{and} \qquad p = q + x(y).[y = z]\tau.\mathbf{0}.$$

Intuitively, p and q are early bisimilar because p can trivially mimic all transitions of q and q can mimic input transitions of the further addend of p because when y is substituted for z, we can choose the transition $q \xrightarrow{xz} \tau.nil$, otherwise, we choose $q \xrightarrow{xw} \mathbf{0}$ for the remaining transitions. In other terms, relation

$$\mathcal{R}_e = \{(p, q), ([z = z]\tau.\mathbf{0}, \tau.\mathbf{0})\} \cup \{([y = z]\tau.\mathbf{0}, \mathbf{0}) : y \neq z\}$$

is an early bisimulation that relates p and q.

On the other hand, p and q are not *late bisimilar because transition $p \xrightarrow{x(y)} [y = z]\tau.\mathbf{0}$ cannot be matched by q. Indeed, according to the late semantics, there are only two possible transitions that we might choose either $q \xrightarrow{x(y)} \mathbf{0}$ or $q \xrightarrow{x(y)} \tau.\mathbf{0}$. In the former case, Definition 2.2 requires that $([y = z]\tau.\mathbf{0})[^w/_y]$ and $\mathbf{0}[^w/_y]$ must be bisimilar for any w, which is not the case when w is z; whereas, in the latter case $([y = z]\tau.\mathbf{0})[^w/_y]$ is bisimilar to $\tau.nil[^w/_y]$ only when $w = z$.*

2.4 Variants of π-calculus

Several variants of the π-calculus have been proposed to study many aspects of concurrent and distributed systems. Since π-calculus has been widely used for modeling many facets of concurrent and distributed computation, pretending to give a complete list of citations would be too much ambitious. We focus here on some variants of π-calculus that are more closely related to our dissertation.

Some presentations of the calculus adopt *replication* [23], usually written as $!p$, in place of recursion. Process $!p$ can be intuitively explained as infinite copies of p in parallel. As far as expressiveness is concerned, the two methods for expressing infinite behaviours are equivalent (at the cost of additional silent actions). However, replication complicates the identification of a syntactic class of *finitary* agents. An agent is finitary if the number of parallel components of all its derivatives is bound. It is not decidable whether an agent is finitary or not, but it is possible to find syntactic conditions that ensures it. In the case of π-calculus with recursion, agents that do not have parallel composition in recursive definitions are finitary. Those agents are called *finite control agents* [8].

The *asynchronous* π-calculus [17, 5, 4] is a simple variant of the π-calculus where asynchrony is achieved by imposing the void continuation to the output actions. In other words, the (synchronous) π-calculus uses both input and output actions as prefixes while its asynchronous counterpart does not allow output prefixes: Outputs are processes of the form $\overline{x}y.\mathbf{0}$. Although from a theoretical point of view asynchronous π-calculus is less expressive than its synchronous version [30], it is still enough expressive in practice [17], and, in many respects, more adequate for modeling distributed computing.

The `join-calculus` [13] is an "extended subset" of asynchronous π-calculus which combines the three operators for input, restriction and replication into a single operator, called *definition*, that has the additional capability of describing atomic *joint* reception of values from different communication channels. The Distributed `join-calculus` [14] adds abstractions to express process distribution and process mobility.

Another linguistic extension is the introduction of polyadicity introduced in [23]. The polyadic version of the allows one to send and receive tuples of names instead of one single name along channels.

A significant variation of polyadic π-calculus is constituted by the *Fusion Calculus* introduced in [31]. The interesting aspect of Fusion Calculus is the complete symmetry between input and output actions. Indeed, input prefixes do not bind their object names. The effect of the synchronization of complementary actions is that object names are (globally) identified as shown in the communication rule below

$$\frac{p \xrightarrow{x\tilde{y}} p' \qquad q \xrightarrow{\overline{x}\tilde{z}} q'}{p \mid q \xrightarrow{[\tilde{y}=\tilde{z}]} p' \mid q'}$$

where $[\tilde{y} = \tilde{z}]$ stands for $[y_1 = z_1, ..., y_n = z_n]$.

Another extension of the polyadic calculus is *Distributed π-calculus* [16, 34]. This extension defines an explicit notion of locality that also affects channels that are *allocated*. Distributed π-calculus models distributed computations and access control policies. Locations reflect the idea of having administrative domains and located channels can be thought of as channels under the control of certain authorities. Moreover, distributed π-calculus provides a form of process mobility because processes can move from through localities.

3 Categories and Functors

It is quite common to consider concurrent and distributed systems as *reactive*, namely as systems which are plugged and executed into an environment that can interact with them by means of some *stimuli* to which systems react. In this context the behaviour of a system can be represented as the ability of the system of reacting to a given class of stimuli. Hence, a natural question is: when two systems have equivalent behaviours? The ability to answer this question is quite important. For instance, it implies that a system S can be unplugged

and substituted with a system S', provided that S' is equivalent to S, namely, provided that the rest of the environment cannot distinguish the behaviour of S' from the behaviour of S. Another reason is related to "efficiency". We can replace S with a "smaller" system S', provided that they are equivalent. Among the wide number of theories for representing systems and their behaviours that have been proposed, π-calculus and *bisimulation* equivalences probably are the most famous and applied.

A very natural and elegant way of describing transitions systems is provided by *coalgebras*, that are the dual concept of algebra. Duality between algebras and coalgebra can be precisely stated in a categorical setting. This section aims at formally reviewing elementary notions of coalgebras. Indeed, we recap only minimal notions necessary for presenting the coalgebraic version of HD-automata. The reader can skip this section if (s)he is already acquainted with the notions of category, functor and co-algebra and with the elementary (polynomial) functors over **Set**. The interested reader is referred to [18, 1] for a deeper study of coalgebras.

We first introduce the concept of *category*.

Definition 3.1 (Category) *A* category $\mathbf{C}$ *is class of* objects $O_{\mathbf{C}}$ *(ranged over by a, b ,...) together with a class of* arrows $A_{\mathbf{C}}$ *(ranged over by by f, g ,...) such that the following properties hold:*

- *Each arrow f has a* domain $dom(f)$ *(also called* source *and a* codomain $cod(f)$ *(also called* target*) which are objects. We write $f : a \to b$ when f is an arrow whose domain is a and whose codomain is b.*
- *Given two arrows f and g such that $cod(f) = dom(g)$, the* composition *of f and g, written $f;g$, is an arrow with domain $dom(f)$ and codomain $cod(g)$.*
- *composition is associative, namely, whenever f, g and h can be composed, $f;(g;h) = (f;g);h$.*
- *For any object a there is an* identity arrow $id_a : a \to a$. *All identity arrows enjoy the following properties:*

$$id_{dom(f)}; f = f = f; id_{cod(f)}.$$

Essentially, a category is a collection of objects having a given structure and a collection of transformations of objects that preserve the structure of objects. We avoid the details of the general theory of category and we limit our presentation to the restricted setting of sets and functions among them.

Observation 3.1 *The category of sets, denoted by* **Set***, is the category having sets as objects and (total) function on sets as arrows. Domain and codomain are the domain and codomain of a function, composition of arrows is the usual function composition, while identities are the identity functions. It is a simple exercise to show that* **Set** *is an instance of Definition 3.1.*

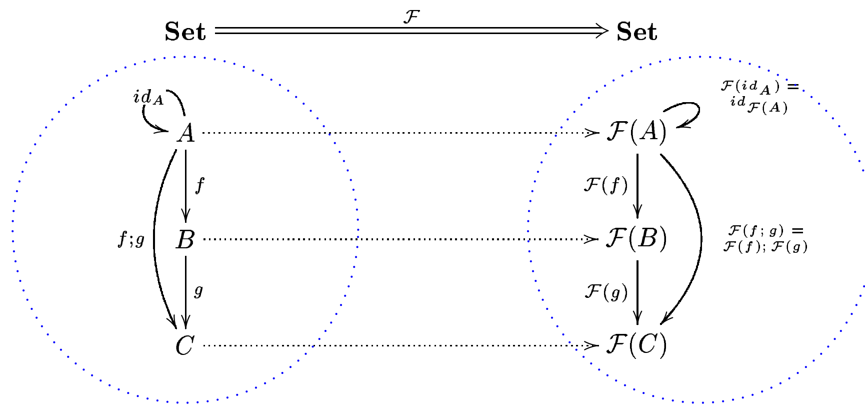


Figure 2: Functor over **Set**

The most important concept from category theory that we need is *functoriality*. Informally, an operation on sets is "functorial" when it can be lifted to functions preserving function composition and identities. Functoriality is a familiar concept in many fields of computer science.

Example 3.1 *Let $L : A \to A^*$ be the function that associates to a set A, the set of the finite lists over A, given a function $f : A \to B$, we can define $L(f) : L(A) \to L(B)$ as the function such that*

$$L(f) : [e_1, ..., e_n] \mapsto [f(e_1), ..., f(e_n)].$$

Notice that $L(f)$ is the usual map operation on lists exploited in functional programming. It is easy to prove that L is functorial, indeed, $L(f;g) = L(f);L(g)$ (if f and g can be composed) and that $L(id_A) = id_{L(A)}$.

The following definition formalizes this concept:

Definition 3.2 (Functor over Set) *An* (endo-)functor $\mathcal{F}$ *over* **Set** *maps sets to sets and functions to functions such that*

- *for each function $f : A \to B$, $\mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(A)$;*
- *for each set A, $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$;*
- *for all composable functions $f : A \to B$ and $g : B \to C$, $\mathcal{F}(f;g) = \mathcal{F}(f);\mathcal{F}(g)$.*

Figure 2 gives a graphical representation of how a functor acts on objects and arrows. In particular, the figure shows how relations among objects and arrows of the starting category are maintained in the target category.

Observation 3.2 *The general definition of functor is given for any two categories* $\mathbf{C}$ *and* $\mathbf{C}'$*. Figure 2 remains the same also in the general case apart that* **Set** *is substituted by* $\mathbf{C}$ *on the left and with* $\mathbf{C}'$ *on the right of* $\mathcal{F}$*.*

By simply applying Definition 3.2 it is possible to show that the identity mapping of sets and functions, or the mapping that associates a constant set L to any set A are functors overs **Set**. Other examples of functor over sets can be given.

Example 3.2 *This example defines two of most useful functors over* **Set***, namely* product *and* co-product *(or disjoint union).*

Let $A \times B$ *the cartesian product of sets* A *and* B*. It is possible to define two functions* $\pi : A \times B \to A$ *and* $\pi' : A \times B \to B$ *that behaves as the* projection *function, i.e.*

$$\pi : (a, b) \mapsto a \qquad\qquad \pi' : (a, b) \mapsto b.$$

Given two functions $f : Z \to A$ *and* $g : Z \to B$*, there exists a unique* pair function $\langle f, g\rangle : Z \to A \times B$ *such that the following equalities hold:*

$$\langle f, g\rangle; \pi = f \qquad\qquad \langle f, g\rangle; \pi' = g.$$

It is worth to give a graphical representation of the above relations between π*,* π'*,* f*,* g *and* $\langle f, g\rangle$*. More precisely, such relations express that the following diagram "commutes"*

$$\begin{array}{ccccc} & & Z & & \\ & \swarrow^{f} & \downarrow{\scriptstyle \langle f,g\rangle} & \searrow^{g} & \\ A & \xleftarrow{\pi} & A \times B & \xrightarrow[\pi']{} & B \end{array}$$

Commutativity of the diagram means that any two paths starting from the same vertex and ending in the same vertex are equal if interpreted as composition of the arrows composing the paths. Moreover, observe that $\langle \pi, \pi'\rangle = id_{A\times B}$ *and that* $h; \langle f, g\rangle = \langle h; f, h; g\rangle$ *for any function* h *such that* $cod(h) = Z$*.*
We can lift the cartesian product to functions. Indeed, if $f : A \to B$ *and* $f' : A' \to B'$*, we can define* $f \times f' : A \times A' \to B \times B'$ *as the function* $\langle \pi; f, \pi'; f'\rangle$*, namely, the function that maps* (a, a') *into* $(f(a), f'(a'))$*. It is easy to verify that the product is functorial, in other words,the following equalities hold:*

$$id_A \times id_{A'} = id_{A\times A'} \qquad\qquad (f; g) \times (f'; g') = (f \times f'); (g \times g').$$

Let $A + B$ *denotes the disjoint union of sets* A *and* B*:*

$$A + B \stackrel{\text{def}}{=} \{0\} \times A \,\cup\, \{1\} \times B.$$

In some sense, disjoint union is the dual of product, from which its synonym co-product *derives: Instead of projections, we can define* co-projections $\kappa : A \to A + B$ *and* $\kappa' : B \to A + B$ *which are defined by*

$$\kappa : a \mapsto (0, a) \qquad\qquad \kappa' : b \mapsto (1, b).$$

Informally, κ and κ' inject elements of A and B (respectively) into $A + B$, while, on the contrary, projections π and π' "extract" elements of A and B (resp.) from $A \times B$. Moreover, analogously to what is done for products, given functions $f : A \to Z$ and $g : B \to Z$, we can built the "co-product *function* $[f, g] : A+B \to Z$ *as the unique function such that* $\kappa; [f, g] = f$ *and* $\kappa'; [f, g] = g$. *It is possible to define $[f, g]$ by case:*

$$[f, g](x) = \begin{cases} f(a), \textit{if } x = (0, a) \\ g(b), \textit{if } x = (1, b) \end{cases}$$

Finally, we lift the co-product to functions by defining $f + g = [f; \kappa, g; \kappa']$. Observe $+$ on functions preserves identities and compositions, i.e. it is a functor.

Another functor that will be very important in defining co-algebras is the *powerset functor*.

Example 3.3 *Let us consider the operation $A \mapsto \wp(A)$, i.e. the function that associates to a set the set of all its subsets and, for a function $f : A \to B$, let us consider*

$$\wp(f) : \wp(A) \to \wp(B) \qquad\qquad \wp(f) : U \mapsto \{f(u) \mid u \in U\}.$$

Then, by definition,

- $\wp(id_A)(U) = \{id_A(u) \mid u \in U\}$, *for any $U \subseteq A$ hence,* $\wp(id_A)(U) = U$;
- $\wp(f; g)(U) = \{g(f(u)) \mid u \in U\}$, *for any $U \subseteq dom(f)$, hence, by definition,* $\wp(f; g)(U) = \wp(g)(\wp(f)(U))$, *for all $U \subseteq dom(f)$ which amounts to* $\wp(f; g) = \wp(f); \wp(g)$.

This proves that the powerset operation is functorial.

We conclude this section by claiming that the above functors are part of the so called polynomial functors that are those functors that can be obtained by combining sums, products, powerset, constant and exponentiation functors. Indeed, it is possible to prove that any combination of this functors is a functor (in fact categories and functors among them form a category) [39, 3].

4 Algebras and coalgebras

Before introducing coalgebras, we show how it is possible to rephrase the more familiar concept of algebra into a categorical framework. This approach also permits us to state the duality of algebras and coalgebras.

Given a signature Σ that, for the sake of simplicity we consider one sorted. We can easily define a Σ-algebra, namely a structure over a given set A that associates functions to any symbol in Σ. The only constraint being the fact that such functions must preserve arity of operations[3].

[3] In the general case of multisorted signatures, also sorting must be preserved.

Example 4.1 *If* $\Sigma = \langle \mathtt{nat}, \mathtt{z} :\to \mathtt{nat}, \mathtt{s} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat} \rangle$*, then a* Σ*-algebra is the algebra of natural numbers: Namely,* nat *is interpreted as the set of natural numbers* ω*, the element* 0 *interprets constant* z *and sum function interprets* s*.*

Referring to Example 4.1, signature Σ "resembles" the functor

$$\mathcal{N}(X) = \mathbf{1} + X \times X,$$

where $\mathbf{1}$ is a singleton set. A $\mathcal{N}$-algebra is a pair (A, α) where A is a set and $\alpha : \mathcal{N}(A) \to A$ is a function that given a set A either returns the element in $\mathbf{1}$ or a "new" element built out of two elements in A.

More generally, if $\Sigma = \{\sigma_1, ..., \sigma_s\}$ is a signature such that each operation σ_i has arity n_i, we can associate a functor

$$\mathcal{F}_\Sigma(U) = U^{n_1} + ... + U^{n_s}$$

(where U^0 is a singleton set containing an element of U) such that a Σ-algebra with carrier A, can be represented by a function $\mathcal{F}_\Sigma(A) \to A$.

We have now all the ingredients for defining coalgebras and point out their duality with respect to algebras. We restrict our definition to coalgebras over endo-functors of **Set**.

Definition 4.1 ($\mathcal{F}$-coalgebra) *Let* $\mathcal{F}$ *be an endo-functor on the category* **Set***. A* $\mathcal{F}$-coalgebra *consists of a pair* (A, α) *such that* $\alpha : A \to \mathcal{F}(A)$*.*

This definition makes clear also the duality between $\mathcal{F}$-algebras and $\mathcal{F}$-coalgebras. Indeed they are functions whose domain and codomain are "reversed", namely, are arrows between the same objects but with opposite directions. Different directions can be interpreted as "construction" and "observation". An $\mathcal{F}$-algebra with carrier set A is a function $\mathcal{F}(A) \to A$ and says how to "construct" elements of A by applying operations detailed by $\mathcal{F}$. On the other hand, a $\mathcal{F}$-coalgebra is a function $A \to \mathcal{F}(A)$ which, given an element of A, returns informations on the element. For instance let us consider $\mathcal{T}(X) = L \times X$, where L is a fixed set, then the coalgebra $\alpha : Q \to L \times Q$ can be though of as an automaton such that, for each state $q \in Q$, if $\alpha(q) = (l, q')$ then q' is the successor state of q reached with a transition labeled l.

5 Transition Systems as Coalgebras

This section gives the preliminary definitions and notations on automata. We present a formal framework that, starting from ordinary automata, introduces the coalgebraic version of automata theory and the coalgebraic definition of HD-automata. Essentially we report here definitions and notations from [11].

In the following we will use terms 'automaton' and 'transition system' interchangeably.

Definition 5.1 (Automata) *An* automaton A *is a triple* $(S, L, \to)$ *where* S *is the set of* states*,* L *the set of* actions *or* labels *and* $\to \subseteq S \times L \times S$ *is the*

transition relation. *Usually, one writes* $s \stackrel{\ell}{\rightarrow} d$ *to indicate* $(s, \ell, d) \in \rightarrow$; *s is the* source *state and d is the* destination *or* target *state. Transition* $s \stackrel{\ell}{\rightarrow} d$ *is also called* 'arrow'.

Observation 5.1 *Classical automata theory an initial state* $\bar{s} \in S$ *is specified for automata. For the moment, we ignore initial states that will be specified when necessary.*

Depending on the transition relation, we can distinguish various classes of automata. For instance, *deterministic* automata are those automata having a transition relation which is functional, i.e. $s \stackrel{\ell}{\rightarrow} d$ and $s \stackrel{\ell}{\rightarrow} d'$ if, and only if, $d = d'$. Deterministic automata have one possible successor state for each state s and each label ℓ. *Non-deterministic automata* are automata which admit more than one possible successor for a state and a label.

We aim at developing a coalgebraic description of the minimization procedure, hence, we rephrase coalgebras in terms of structures that are more concrete than functors. Indeed, we provide a (concrete) representation of the terminal coalgebra (of an endofunctor over `Set`) in terms of sets and *quadruples* which will yield the minimal transition system. In particular, we define *bundles* as the concrete structures that are associated by the co-algebraic functor to states that will be (concretely) represented as objects of `Set`. In this way it is possible to express the functional aspect of the functor (at each step of the minimization algorithm) by means of particular structures that will be introduced in the following.

Before introducing bundles it is worth to give some notations that are hereafter used:

- Q : `Set` denotes a set and $q : Q$ denotes an element in the set Q;
- `Fun` is the collection of functions among sets (the arrows of category **Set**). The function space over sets will have the following structure:

$$\texttt{Fun} = \{H \mid H = \langle S : \texttt{Set}, D : \texttt{Set}, h : S \rightarrow D \rangle\}.$$

  By convention we use S_H, D_H and h_H to respectively denote domain, codomain and mapping of an element of `Fun`.

Let H and K be functions (i.e. elements of `Fun`), then the *composition* of H and K $(H; K)$ is defined provided that $S_K = D_H$ and it is the function given by $S_{H;K} = S_H$, $D_{H;K} = D_K$, and $h_{H;K} = h_K \circ h_H$. Sometimes, we shall need to work with surjective functions. Hence we let $\widehat{H}$ be the function given by $S_{\widehat{H}} = S_H$, $D_{\widehat{H}} = \{q' : D_H \mid \exists q : S_H, h_H(q) = q'\}$ and $h_{\widehat{H}} = h_H$, where H is a function. Hereafter, $h : A \stackrel{bij}{\rightarrow} B$ $(h : A \stackrel{inj}{\rightarrow} B)$ denotes a bijective (injective) function from A to B.

A finite-state transition system can be coalgebraically described by employing two ingredients: A set Q, that represents the state space, together with a function $K : Q \rightarrow \wp_{\mathrm{fin}}(L \times Q)$ that represents the "behaviour" of the transition system: $K(q)$ is the set of pairs (ℓ, q') such that $q \stackrel{\ell}{\rightarrow} q'$.

Definition 5.2 (Bundles) *Let L be the set of labels (ranged over by ℓ), then a* bundle *β over L is a structure $\langle D : \texttt{Set}, Step : \wp_{\mathrm{fin}}(L \times D)\rangle$. We call the first component of a bundle β* the support *of β. Given a fixed set of labels L, by convention, B^L denotes the collection of bundles and $\beta : B^L$ means that β is a bundle over L.*

Intuitively, the notion of bundle has to be understood as giving the data structure representing all the state transitions out of a given state. It details which states are reachable by performing certain actions.

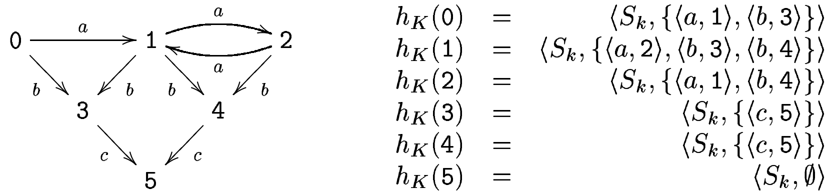
Once a set of labels L has been fixed, we can consider the polynomial endofunctor $\mathcal{A}(X) = \wp_{\mathrm{fin}}(L \times X)$ in $\texttt{Set}$. Functor $\mathcal{A}$ operates on both sets and functions, and characterizes a whole category of labeled transition systems, i.e. of coalgebras. The following clauses define $\mathcal{A}$.

- $\mathcal{A}(Q) = \{\beta : B^L \mid D_\beta = Q\}$, for each $Q : \texttt{Set}$;
- For each $H : \texttt{Fun}$, $\mathcal{A}(H)$ is defined as follows:
  - $S_{\mathcal{A}(H)} = \mathcal{A}(S_H)$ and $D_{\mathcal{A}(H)} = \mathcal{A}(D_H)$;
  - $h_{\mathcal{A}(H)}(\beta : \mathcal{A}(S_H)) = \langle D_H, \{\langle \ell, h_H(q)\rangle \mid \langle \ell, q\rangle : Step_\beta\}\rangle$.

Definition 5.3 (Transition systems as coalgebras) *Let L be a set of labels. Then a labeled transition system over L is a coalgebra for functor $\mathcal{A}$, namely it is a function K such that $D_K = \mathcal{A}(S_K)$.*

Note that the convention of functions allows us not to mention the carrier of a coalgebra K it is implicitly given by S_K.

Example 5.1 *A coalgebra K for functor $\mathcal{A}$ represents a transition system where S_K is the set of states, and $h_K(q) = \beta$, with $D_\beta = S_K$. Let us consider a finite-state automaton and its coalgebraic formulation via the mapping h_K.*



$$\begin{aligned}
h_K(0) &= \langle S_k, \{\langle a, 1\rangle, \langle b, 3\rangle\}\rangle \\
h_K(1) &= \langle S_k, \{\langle a, 2\rangle, \langle b, 3\rangle, \langle b, 4\rangle\}\rangle \\
h_K(2) &= \langle S_k, \{\langle a, 1\rangle, \langle b, 4\rangle\}\rangle \\
h_K(3) &= \langle S_k, \{\langle c, 5\rangle\}\rangle \\
h_K(4) &= \langle S_k, \{\langle c, 5\rangle\}\rangle \\
h_K(5) &= \langle S_k, \emptyset\rangle
\end{aligned}$$

Note how, for each state $q \in \{0, ..., 5\}$, $h_K(q)$ yields all the immediate successor states of q and the corresponding labels. In other words, $(\ell, q') \in Step_{h_K(q)}$ if, and only if, $q \xrightarrow{\ell} q'$.

We can rephrase the concepts of coalgebras homomorphism and finality for transition systems:

Definition 5.4 (Homomorphism and finality of transition systems) *Let K and F be two transition systems. A function H is a* homomorphism *of transition system if*

$$S_H = S_K, \qquad D_H = S_F, \qquad H; F = K; \mathcal{A}(H).$$

Which can be represented by the following commuting diagram:

$$\begin{array}{ccc} S_K & \xrightarrow{h_H} & S_F \\ \downarrow{\scriptstyle K} & & \downarrow{\scriptstyle F} \\ \mathcal{A}(S_K) & \xrightarrow[\mathcal{A}(h_H)]{} & \mathcal{A}(S_F) \end{array}$$

A transition system F is final *if, for any other transition system K, there is a unique homomorphism from K to F.*

As usual, homomorphisms correspond to the idea of functions which commutes with the coalgebraic operations while finality encompass the idea of minimality. Indeed, final transition systems are those transition systems that are image of all other transition systems in a certain class through functions which preserves their "behaviour". General results (e.g. [2]) ensure the existence of the final coalgebra for a large class of functors. These results apply to formulation of transition systems in Definition 5.3. In particular, it is interesting to see the result of the iteration along the terminal sequence [39] of functor $\mathcal{A}$.

Let K be a transition system, and let $H_0, H_1, \ldots, H_{i+1}, \ldots$ be the sequence of functions computed by $H_{i+1} = \widehat{K; \mathcal{A}(H_i)}$, where H_0 is the unique function from S_K to the one-element set $\{*\}$ given by $S_{H_0} = S_K$; $D_{H_0} = \{*\}$; and $h_{H_0}(q : S_{H_0}) = *$. Finiteness of $\wp_{\text{fin}}$ ensures convergence of the iteration along the terminal sequence. In [11], the following result is stated:

Theorem 5.1 *Let K be a finite-state transition system. Then,*

- *The iteration along the terminal sequence converges in a finite number of steps, i.e. $D_{H_{i+1}} \equiv D_{H_i}$,*
- *The isomorphism mapping $F : D_{H_i} \rightarrow D_{H_{i+1}}$ yields the minimal realization of transition system K.*

6 A comparison

Comparing the coalgebraic construction with the standard algorithm [20, 9] which constructs the minimal labeled transition system we can observe:

- at each iteration i the elements of D_{H_i} are the blocks of the minimization algorithm (i.e. the i-th partition). Notice that the initial approximation D_{H_0} contains a single block: in fact H_0 maps all the states of the transition system into $\{*\}$.
- at each step the algorithm creates a new partition by identifying the *splitters* for states q and q'. This corresponds in our coalgebraic setting to the fact that $H_i(q) = H_i(q')$ but $H_{i+1}(q) \neq H_{i+1}(q')$.

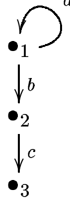


Figure 3: Minimal labeled transition system

- the iteration proceeds until a stable partition of blocks is reached: then the iteration along the terminal sequence converges.

We now apply the iteration along the terminal sequence to the coalgebraic formulation of the transition system of Example 5.1. The initial approximation is the function H_0 defined as follows

$$H_0 = \langle S_{H_0} = S_K, D_{H_0} = \{*\}, h_{H_0} : q \mapsto *\rangle$$

We now construct the first approximation H_1. We have that

$$h_{H_1} : q \mapsto \langle D_{H_0}, \{\langle \ell, h_{H_0}(q')\rangle : q \xrightarrow{\ell} q'\}\rangle$$

In our example we obtain the function h_{H_1} and the destination state $D_{H_1} = \{\beta_1, \beta_2, \beta_3\}$ as detailed below.

$$\begin{array}{rcr} h_{H_1}(0) & = & \langle\{*\}, \{\langle a, *\rangle, \langle b, *\rangle\}\rangle \\ h_{H_1}(1) & = & \langle\{*\}, \{\langle a, *\rangle, \langle b, *\rangle\}\rangle \\ h_{H_2}(2) & = & \langle\{*\}, \{\langle a, *\rangle, \langle b, *\rangle\}\rangle \\ h_{H_1}(3) & = & \langle\{*\}, \{\langle c, *\rangle\}\rangle \\ h_{H_1}(4) & = & \langle\{*\}, \{\langle c, *\rangle\}\rangle \\ h_{H_1}(5) & = & \langle\{*\}, \emptyset\rangle \end{array} \qquad \begin{array}{rcr} \beta_1 & = & \langle\{*\}, \{\langle a, *\rangle, \langle b, *\rangle\}\rangle \\ \beta_2 & = & \langle\{*\}, \{\langle c, *\rangle\}\rangle \\ \beta_3 & = & \langle\{*\}, \emptyset\rangle \end{array}$$

We continue to apply the iterative construction, we obtain:

$$\begin{array}{rcr} h_{H_2}(0) & = & \langle D_{H_1}, \{\langle a, \beta_1\rangle, \langle b, \beta_2\rangle\}\rangle \\ h_{H_2}(1) & = & \langle D_{H_1}, \{\langle a, \beta_1\rangle, \langle b, \beta_2\rangle\}\rangle \\ h_{H_2}(2) & = & \langle D_{H_1}, \{\langle a, \beta_1\rangle, \langle b, \beta_2\rangle\}\rangle \\ h_{H_2}(3) & = & \langle D_{H_1}, \{\langle c, \beta_3\rangle\}\rangle \\ h_{H_2}(4) & = & \langle D_{H_1}, \{\langle c, \beta_3\rangle\}\rangle \\ h_{H_2}(5) & = & \langle D_{H_1}, \beta_3\rangle \end{array}$$

Since $D_{H_2} \equiv D_{H_1}$ the iterative construction converges, thus providing the minimal labeled transition system illustrated in Figure 3, where $\bullet_1 = \{0, 1, 2\}$, $\bullet_2 = \{3, 4\}$ and $\bullet_3 = \{5\}$.

7 HD-automata for π-agents

Names appear explicitly in the states of an HD-automaton: the idea is that the names associated to a state are those names which may play a rôle in the state evolution. Let $\mathcal{N}$ be an infinite countable set of names ranged over by v and let $\mathcal{N}^\star$ be the set $\mathcal{N} \cup *$, where $* \notin \mathcal{N}$ is a distinguished name and will be used for modeling name creation. We also assume that $<$ is a total order on $\mathcal{N}^\star$ (for instance, it can be the lexicographic order on $\mathcal{N}$ and $\forall v \in \mathcal{N} : * < v$). Given a state q of a HD-automaton, a set $\{v_1, \ldots, v_{|q|}\} \subseteq \mathcal{N}$ of local names and a permutation group G_q are associated with q. Elements of G_q are those permutations of names of q that leave unchanged the behaviour of q. Moreover, the identity of names is local to the state: States which differ only for the order of their names are identified. Due to the usage of local names, whenever a transition is performed a name correspondence between the name of the source state and the names of the target state is explicitly required.

Now we introduce the notion of *named sets*.

Definition 7.1 (Named set) *A* named set *A is a structure*

$$A = \langle Q : \mathit{Set}, |_| : Q \longrightarrow \omega, \leq : \wp(Q \times Q), G : \prod_{q \in Q} \wp(\{v_1..v_{|q|}\} \xrightarrow{bij} \{v_1..v_{|q|}\})\rangle$$

where $\forall q : Q_A$, $G_A(q)$ *is a permutation group and* $\leq_A$ *is a total ordering.*

A named set represents a set of states equipped with a mechanism to give local meaning to names occurring in each state. In particular, function $|_|$ yields the number of local names of states. Moreover, the permutation group $G_A(q)$ allows one to describe directly the renamings that do not affect the behaviour of q, i.e., symmetries on the local names of q. Finally, we assume that states are totally ordered. By convention we write $\{q : Q_A\}_A$ to indicate the set $\{v_1, ..., v_{|q|_A}\}$ and we use `NSet` to denote the universe of named sets.

In Definition 7.1 and in the following, the general product $\prod$ is employed (as usual in type theory) to type functions f such that the type of $f(q)$ is dependent on q.

Definition 7.2 (Named function) *A* named function *H is a structure*

$$H = \langle S : \mathit{NSet}, D : \mathit{NSet}, h : Q_S \longrightarrow Q_D, \Sigma : Q_S \longrightarrow \wp(\{h(q)\}_D \xrightarrow{inj} \{q\}_S)\rangle$$

where $\forall q : Q_{S_H}, \forall \sigma : \Sigma_H(q)$,

1. $G_{D_H}(h_H(q)); \sigma = \Sigma_H(q)$ *and*
2. $\sigma; G_{S_H}(q) \subseteq \Sigma_H(q)$.

As in the case of standard transition systems, functions are used to determine the possible transitions of a given state. The idea is that for each state q in S_H, $h_H(q)$ yields the behaviour of q, i.e. the transitions departing from q. Since states are equipped with local names, a name correspondence (the mapping H_h)

is needed to describe how names in the destination state are mapped into names of the source state, therefore we must equip H with a *set* $\Sigma_H(q)$ of injective functions. However, names of corresponding states $(q, h_H(q))$ in h_H are defined up to permutation groups and name correspondence must not be "sensible" to the local meaning of names, therefore, the whole set $\Sigma_H(q)$ must be generated by saturating any of its elements by the permutation group of $h_H(q)$, and the result must be invariant with respect to the permutation group of q. Condition (1) in Definition 7.2 states that the group of $h_H(q)$ does not change the meaning of names in $h_H(q)$, while Condition (2) states that the group of q does not "generate meanings" for local names of q that are outside $h_H(q)$.

Named functions can be composed in the obvious way. Let H and K be named functions. Then $H;K$ is defined only if $D_H = S_K$, and

$$S_{H;K} = S_H, \qquad D_{H;K} = D_K, \qquad h_{H;K} : Q_{S_H} \longrightarrow Q_{D_K} = h_H; h_K,$$
$$\Sigma_{H;K}(q : Q_{S_H}) = \Sigma_K(h_H(q)); \Sigma_H(q)$$

Let H be a named function, $\widehat{H}$ denotes the surjective component of H:

- $S_{\widehat{H}} = S_H$ and $Q_{D_{\widehat{H}}} = \{q' : Q_{D_H} \mid \exists q : Q_{S_H}.h_H(q) = q'\}$,
- $|q|_{D_{\widehat{H}}} = |q|_{D_H}$,
- $G_{D_{\widehat{H}}}(q) = G_{D_H}(q)$,
- $h_{\widehat{H}}(q) = h_H(q)$,
- $\Sigma_{\widehat{H}}(q) = \Sigma_H(q)$

7.1 Bundles over π-calculus actions

We want to represent the transition system for the early semantics of π-calculus reported in Section 2. The notion of bundle must be enriched. First we have to fix the set of labels of transitions. Labels of transitions must distinguish among the different meanings of names occurring in π-calculus actions, namely synchronization, bound/free output and bound/free input.

The set of π-calculus labels L_π is the set $\{TAU, BOUT, OUT, BIN, IN\}$. We specify two different labels for input actions: Label BIN is used when the input transition acquires a new name, namely a name that was not previously known to the agent, while IN corresponds to an input transition that acquires an already known name.

Since names are local to states, it is necessary to specify how label names are related to names of states. For instance, no name is associated to synchronization labels, whereas one name, is associated to bound output labels. Let $|_|$ be the weight map associating to each π-label the set of indexes of distinct names the label refers to. The weight map is defined as follows:

$$|TAU| = \emptyset \quad |BOUT| = |BIN| = \{1\} \quad |OUT| = |IN| = \{1, 2\}$$

Definition 7.3 (Bundles) *A* bundle β *consists of the structure*

$$\beta = \langle\ \mathcal{D} : \texttt{NSet}, Step : \wp(qd\,\mathcal{D})\ \rangle$$

where $qd\,\mathcal{D}$ *is the set of* quadruples *of the form* $\langle \ell, \pi, \sigma, q\rangle$ *given by*

$$qd\,\mathcal{D} = \{\langle \ell : L_\pi, \pi : |\ell| \xrightarrow{inj} \{v_1..\}, \sigma : \prod_{\ell \in L_\pi} \{q\}_\mathcal{D} \xrightarrow{inj} Q\ell, q : Q_\mathcal{D}\rangle\}.$$

and

$$Q\ell = \begin{cases} \{*, v_1, ...\} & \text{if } \ell \in \{BOUT, BIN\} \\ \{v_1, ...\} & \text{if } \ell \notin \{BOUT, BIN\} \end{cases}$$

under the constraint that $G_{\mathcal{D}_\beta}(q); S_q = S_q$*, where* $S_q = \{\langle \ell, \pi, \sigma, q\rangle \in Step_\beta\}$ *and* $\rho; \langle \ell, \pi, \sigma, q\rangle = \langle \ell, \pi, \rho; \sigma, q\rangle$.

As above, the intuition is that the $Step$ component of a bundle describes the set of successor states for a given source state. More precisely, if $\langle \ell, \pi, \sigma, q\rangle \in qd\ \mathcal{D}$, then q is the destination state; ℓ is the label of the transition; π associates to the label the names observed in the transition; and σ states how names in the destination state are related with the names in the source state. Notice that the distinguished element $*$ belongs to the names of the source state when a new name is generated in the transition.

In order to exploit named functions for representing HD-automata it is necessary to equip the set of bundles B with a named set structure. In other words we must define a total order on bundles, a function that maps a bundle to its number of names and a group of permutations over those names.

Definition 7.4 (Bundle names) *Let* β *be a bundle. Function* $\{\!|_|\!\} : B \to \mathcal{N}$*, mapping each bundle to the set of its names, is defined by*

$$\{\!|\beta|\!\} = \bigcup_{\langle \ell, \pi, \sigma, q\rangle \in Step_\beta} rng(\pi) \cup rng(\sigma) \setminus \{*\}$$

where rng *yields the range of functions. We only consider bundles* β *such that* $\{\!|\beta|\!\}$ *is finite and we let* $\lfloor\beta\rfloor$ *to indicate the number of names which occur in the bundle* β *(i.e.* $\lfloor\beta\rfloor = |\{\!|\beta|\!\}|$*).*

The minimization algorithm necessitates of a mechanism for determining the representative element of a given class of equivalent states. Intuitively, two states are equivalent when they have the "same" bundles, hence, the choice of a canonical state turns in the choice of a canonical bundle. We are interested in class of bundles defined on a given set of labels and states. For those bundles we can assume that a total order on states and labels exist. Hence, quadruples are totally ordered, e.g. assuming the lexicographic order of labels, states and names. The order over quadruples yields an ordering $\sqsubseteq$ over bundles. This ordering relation will be used to define canonical representatives of bundles. The ordering on quadruples can be defined non ambiguously only assuming an ordering on bundles support.

Finally, the group of a bundle can be defined once we define how a permutation is applied to a bundle. Given a bundle β and a permutation $\theta : \{\!|\beta|\!\} \xrightarrow{bij} \{\!|\beta|\!\}$, bundle $\beta;\theta$ is defined as $\mathcal{D}_{\beta;\theta} = \mathcal{D}_\beta$, $step_{\beta;\theta} = \{\langle \ell, \pi;\theta, \sigma;\theta, q\rangle \mid \langle \ell, \pi, \sigma, q\rangle : \beta\}$.

The most important construction on bundles is the *normalization* operation. This operation is necessary for two different reasons. The first reason is that there are different equivalent ways for picking up the step components (i.e. quadruples $\langle \ell, \pi, \sigma, q\rangle$) of a bundle.

The second, more important, reason for normalizing a bundle is for removing from the step component of a bundle all the input transitions which are *redundant*. Indeed, redundant transitions occur when an HD-automaton is built from a π-calculus agent. During this phase, it is not possible to decide which free input transitions are required, and which transitions are covered by the bound input transition[4]. The solution to this problem consists of adding a superset of the required free input transitions when the HD-automaton is built, and to exploit a reduction function to remove the ones that are unnecessary. Consider for instance the case of a state q having only one name v_1 and assume that the following two tuples appear in a bundle:

$$\langle IN, xy, \{v_1 \to y\}, q\rangle \qquad \text{and} \qquad \langle BIN, x, \{v_1 \to *\}, q\rangle.$$

Then, the first tuple is redundant, if y is not an active in q as it expresses exactly the same behaviour of the second tuple, except that a "free" input transition is used rather than a "bound" one. Hence, the transformation removes the first tuple from the bundle. During the iterative execution of the minimization algorithm, bundles are split: this means that the set of redundant components of bundles decreases. Hence, when the iterative construction terminates, only those free inputs that are really redundant have been removed from the bundles.

The normalization of a bundle β is done in different steps. First, the bundle is reduced by removing all the possibly redundant input transitions. Reduction function $red(\beta)$ on bundles is defined as follows:

- $\mathcal{D}_{red(\beta)} = \mathcal{D}_\beta$,
- $Step_{red(\beta)} = Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid \langle BIN, x, \sigma', q\rangle : Step_\beta \wedge \sigma' = \sigma; \{y \to *\}\}$.

where $\sigma; \{y \to *\}$ is the function equal to σ on any name different from y and that assigns $*$ to y. Once the redundant input transitions have been removed, it is possible to associate to bundle β the set of its "active names" $an_\beta = \{\!| red(\beta) |\!\}$. These are the names that appear either in a destination state or in a label of a non-redundant transition of the bundle. Finally, the *normalization* function $norm(\beta)$ is defined as follows:

- $\mathcal{D}_{norm(\beta)} = \mathcal{D}_\beta$
- $Step_{norm(\beta)} = min_{\sqsubseteq}\big(Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid y \notin an_\beta\}\big)$,

where $min_{\sqsubseteq}$ is the function that, when applied to $Step_\beta$, returns the step of the minimal bundle (with respect to order $\sqsubseteq$) among those obtained by permuting

[4]In the general case, to decide whether a free input transition is required it is as difficult as to decide the bisimilarity of two π-calculus agents.

names of β in all possible ways. More precisely, given a bundle $\overline{\beta}$, $min_{\sqsubseteq}\overline{\beta}$ is the minimal bundle in $\{\overline{\beta};\theta \mid \theta : \{\!|\overline{\beta}|\!\} \xrightarrow{bij} \{\!|\overline{\beta}|\!\}\}$, with respect to the total ordering $\sqsubseteq$ of bundles over $\mathcal{D}$. The order relation $\sqsubseteq$ is used to define the canonical representatives of bundles and relies on the order of quadruples. For this reason we introduced an ordering relation on named sets in the first place. In the following, we use $perm(\beta)$ to denote the canonical permutation that associates $Step_{norm(\beta)}$ and $Step_\beta \setminus \{\langle IN, xy, \sigma, q\rangle \mid y \notin an_\beta\}$.

We remark that, while *all* IN transitions covered by BIN transitions are removed in the definition of $red(\beta)$, only those corresponding to the reception of non-active names are removed in the definition of $norm(\beta)$. In fact, even if an input transition is redundant, it might be the case that it corresponds to the reception of a name that is active due to some other transitions.

Finally, we need a construction which extracts in a canonical way a group of permutations out of a bundle. Let β be a bundle, define $Gr\ \beta$ to be the set $\{\rho \mid Step_\beta;\rho^* = Step_\beta\}$. Where, given a function f, f^* is its $*$-extension and is defined as:

$$f^*(x) = \begin{cases} * & \text{if } x = * \\ f(x) & \text{otherwise} \end{cases}$$

Proposition 7.1 *$Gr\ \beta$ is a group of permutations.*

7.2 The minimization algorithm

We are now ready to introduce the functor $\mathcal{T}$ that defines the coalgebras for HD-automata. The action of functor $\mathcal{T}$ over named sets is given by:

- $Q_{\mathcal{T}(A)} = \{\beta : Bundle \mid \mathcal{D}_\beta = A, \beta \text{ normalized}\}$,
- $|\beta|_{\mathcal{T}(A)} = \lfloor \beta \rfloor$,
- $G_{\mathcal{T}(A)}(\beta) = Gr\ \beta$,
- $\beta_1 \leq_{\mathcal{T}(A)} \beta_2$ iff $Step_{\beta_1} \sqsubseteq Step_{\beta_2}$,

while the action of functor $\mathcal{T}$ over named functions is given by:

- $S_{\mathcal{T}(H)} = \mathcal{T}(S_H)$, $D_{\mathcal{T}(H)} = \mathcal{T}(D_H)$,
- $h_{\mathcal{T}(H)}(\beta : Q_{\mathcal{T}(S_H)}) : Q_{\mathcal{T}(D_H)} = norm(\beta')$,
- $\Sigma_{\mathcal{T}(H)}(\beta : Q_{\mathcal{T}(S_H)}) = Gr(norm(\beta')); (perm(\beta'))^{-1}; inj : \{\!| norm(\beta') |\!\} \longrightarrow \{\!|\beta|\!\}_{\mathcal{T}(S_H)}$ where $\beta' = \langle D_H, \{\langle \ell, \pi, \sigma'; \sigma, h_H(q)\rangle \mid \langle \ell, \pi, \sigma, q\rangle : Step_\beta, \sigma' : \Sigma_H(q)\}\rangle$.

Notice that functor $\mathcal{T}$ maps every named set A into the named set $\mathcal{T}(A)$ of its *normalized* bundles. Also a named function H is mapped into a named function $\mathcal{T}(H)$ in such a way that every corresponding pair $(q, h_H(q))$ in h_H is mapped into a set of corresponding pairs $(\beta, norm(\beta'))$ of bundles in $h_{\mathcal{T}(H)}$. The quadruples of bundle β' are obtained from those of β by replacing q with $h_H(q)$ and by saturating with respect to the set of name mappings in $\Sigma_H(q)$. The name mappings in $\Sigma_{\mathcal{T}(H)}\beta$ are obtained by transforming the permutation group of bundle $norm(\beta')$ with the inverse of the canonical permutation of β' and with a fixed injective function inj mapping the set of names of $norm(\beta')$ into the set of names of β, defined as $i < j$, $inj(v_i) = v_{i'}$ and $inj(v_j) = v_{j'}$

implies $i' < j'$. Without bundle normalization, the choice of β' among those in $\beta';\theta$ would have been arbitrary and not canonical with the consequence of mapping together fewer bundles than needed.

Definition 7.5 (Transition systems for π-agents) *A transition system over named sets and π-actions is a named function K such that $D_K = \mathcal{T}(S_K)$.*

HD-automata are particular transition systems over named sets. Formally, an HD-automaton A is given by:

- the elements of the state Q_A are π-agents $p(v_1, ..., v_n)$ ordered lexicographically: $p_1 \leq_A p_2$ iff $p_1 \leq_{lex} p_2$
- $|p(v_1, ..., v_n)|_A = n$,
- $G_A q = \{id : \{q\}_A \longrightarrow \{q\}_A\}$, where id denotes the identity function,
- $h : Q_A \longrightarrow \{\beta \mid \mathcal{D}_\beta = A\}$ is such that $\langle \ell, \pi, \sigma, q' \rangle \in Step_{h(q)}$ represent the π-calculus transitions from agent q.

We remark that bundle $Step_{h(q)}$ should not contain all the transitions from q, but only a representative subset. For instance, it is not necessary to consider a free input transition where the received name is not active provided that there is a bound input transition which differs from it only for the bound name. Finally, by using renaming σ in the element of the bundles, it is possible to identify all those π-agents that differ only for an injective renaming. In the following, we represent as $q \xrightarrow{\ell,\pi,\sigma} q'$ the "representative" transitions from agent q that are used in the construction of the HD-automaton.

We can now define the function K.

- $S_K = A$,
- $h_K(q) = norm(h(q))$,
- $\Sigma_K(q) = Gr(h_K(q)); (perm(h(q)))^{-1}; inj : \{\!|\, h(q) \,|\!\} \longrightarrow \{q\}_A$

We now construct the minimal HD-automata by an iterative procedure. We first need to define the initial approximation. Given a HD-automata K, the initial approximation H_0 is defined as follows:

- $S_{H_0} = S_K$, $D_{H_0} = unit$ where $Q_{unit} = \{\star\}$, $|\star|_{unit} = 0$ (and hence $\{\star\} = \phi$), $G_{unit} \star = \phi$, and $\star \leq_{unit} \star$,
- $h_{H_0}(q : Q_{s_{H_0}}) = \star$,
- $\Sigma_{H_0} q = \{\phi\}$

The formula which details the iterative construction is given by

$$H_{i+1} = \widehat{K;\mathcal{T}(H_i)}.$$

Theorem 7.1 *Let K be a finite state HD-automaton. Then*

- *The iteration along the terminal sequence converges in a finite number of steps: n exists such that $D_{H_{i+1}} \equiv D_{H_i}$,*
- *The isomorphism mapping $F : D_{H_i} \to D_{H_{i+1}}$ yields the minimal realization of the transition system K up to strong early bisimilarity.*

The following functional expression (in a extended λ-calculus) makes the iteration step of the normalization algorithm explicit.

$$h_{H_{i+1}} = (\lambda q.norm\ \langle A, \{\langle \ell, \pi, \sigma, q' \rangle \mid q \xrightarrow{\ell,\pi,\sigma} q'\}\rangle);$$
$$\lambda\beta.norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q)\rangle \mid \langle \ell, \pi, \sigma, q\rangle : Step_\beta, \sigma' : \Sigma_{H_i}(q)\}\rangle$$

$$h_{H_{i+1}}(q) = norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q')\rangle \mid q \xrightarrow{\ell,\pi,\sigma} q', \sigma' : \Sigma_{H_i}(q')\}\rangle.$$

Notice that the normalization on the transition system is absorbed by the normalization on the resulting bundle.

Part II

Mihda: A Verification Environment

This part presents Mihda, a verification environment centered around the minimization algorithm presented in Section 7. Mihda is written in ocaml [7] and the most relevant implementation choices are discussed also with respect to the ocaml features. We discuss the architectural aspects of the environment in Section 8, while the principal data structures are detailed in Section 9. Section 10.1 contains the main result of the part which is the correctness of the implementation of the minimization algorithm described in 7. In all these sections the emphasis is on tight connection between the formal co-algebraic specification of the framework and the ocaml implementation which nicely represents the interplay between theory and practice. Indeed, Mihda can be seen as the experimental validation of the theoretical framework proposed in [28, 11], ocaml features interestingly permit to prove correctness with respect to the theoretical framework and the experimental environment also suggests new issues for theoretical investigation.

Mihda can be downloaded from `http://jordie.di.unipi.it:8080/mihda`, where also an interactive interface (detailed in [10]) is available.

8 Architectural Aspects of Mihda

The main features of ocaml exploited in our realization are polymorphism and encapsulation. Polymorphism is one of the intrinsic peculiarity of ML-language family, while encapsulation may be obtained in ocaml in two different ways; the

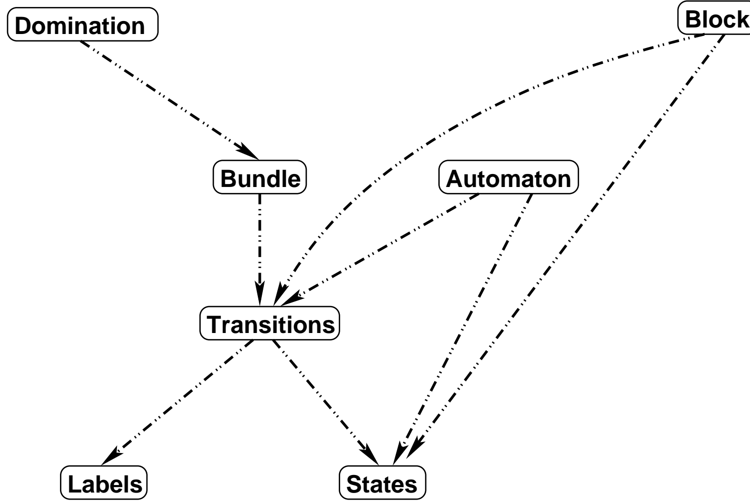


Figure 4: Mihda Architecture

first way is by using the object oriented features of the language, the second way is provided by modular programming features. More precisely, the module system separates the definition of interface specification, called *signatures* (i.e. definition of *abstract data types*) from their realizations, called *structures*. A structure may be parameterized using *functors*. An ocaml functor constructs new modules by mapping modules of a given signature on structures of other signatures.

Observation 8.1 *Object oriented programming simply adds to polymorphism and encapsulation (features already present in functional programming) hierarchical relations among abstract data types. However, in our case, those relations does not play any rôle and, therefore, have not been exploited.*

Language ocaml has been chosen also for other reasons. As detailed in Section 7, the algorithm has been specified in a "type-theoretic" style and the underlying type system makes use of parametric polymorphism. The type system of ocaml offers all the necessary features for handling those kind of type. As a further benefit, Mihda remains tightly close to its specification as we will prove later.

Figure 8 reports (part of) the modules of Mihda and represents the relationships and dependencies among those modules. Nodes of the graph represent both Mihda modules and their principal types, e.g. State is the module for states and contains a declaration State_t of its main type that declares the type of the states of the automata. Two modules in Figure 8 are connected when the upper module declares a type to be the same as the lower module. For instance, the arc connecting Bundle and Transitions states that in module Bundle is declared a type that must be State_t declare in State.

Mihda allows the user to specify automata type and, after having implemented some functionalities on such data structures, a general minimization algorithm is applied and the minimal realization of the automaton is returned. This choice gives the opportunity of applying the same algorithm to different kind of automata that can be defined, provided that they are specified in a manner that respects the constraints imposed by functors. Those constraints expresses equalities that types must satisfy as said before. Notice that this mechanism also aids in implementing different semantic minimization. Let us consider a scenario where, modules of Mihda have been specified for a given class of automata. Later, we decide that the equivalence relation that must be considered should be changed. Conceptually, the algorithm and type declarations for states and automataremain unchanged. Indeed, it would be reasonable to modify only the module Domination (and perhaps Transitions[5]) that is the Mihda module where the (type of) semantic relation is declared. Mihda architecture allows programmer in the scenario depicted above to write new code only for the module Domination.

An easy exercise that we have done is exploit the architecture of Mihda to adapt minimization of HD-automata to minimization of ordinary automata (we refer the interested reader to the web page of the Mihda project for detailed comments).

9 Main data structures

We discuss here the main data structures used in Mihda together with their most important properties. Moreover, their relations with the "theoretical" objects they implement is pointed out.

In the following sections we will use typewriter symbols to denote names for ocaml functions and variables. A list `l` is written as $[\mathtt{e}_1; ...; \mathtt{e}_h]$ while $\mathtt{l}_i$ denotes its i-th element (i.e. $\mathtt{e}_i$). Finally, we write $\mathtt{e} \in \mathtt{l}$ to indicate that `e` is an item of list `l`.

As a general remark, we point out that finite sets will be generally represented as lists. We say that a list `x` corresponds to a finite set X if, and only if, for each element e in X there exists $\mathtt{e} \in \mathtt{x}$ such that `e` *corresponds* to e. Later, we will define various *correspondence* relations over elements which depend on the type they live in. Note that, for each element $e \in X$, many instances of `e` can occur in `x`. However, we will often apply the function `Utils.unique` which removes multiple occurrences of items in a list.

In the following we will exploit two auxiliary functions that are reported here. We also state and prove some properties that will be used in the proof of the correctness of Mihda.

`list_rem el list` returns the list obtained from `list` by removing all the occurrences of items equal to `el`. Its definition is

[5] In general, transition types depend on the semantic relation because the new relation might require information that must be added on the labels.

```
let rec list_rem el = function
  | []      →  []
  | e::es →
    if (compare e el) == 0
    then list_rem el es
    else e :: (list_rem el es)
```

Each element occurring in the list is not removed if different from el. Indeed, the following lemma holds:

Lemma 9.1 $\forall$a.$\forall$b.$\forall$ls $\neq$ b $\wedge$ a $\in$ ls $\Rightarrow$ a $\in$ (Utils.list_rem b ls)

Proof. We proceed by induction on the length of ls. If ls = [] then the implication trivially holds because the antecedent is false. If ls = e :: es then, by definition,

```
if (compare e b) == 0
then list_rem b es
else e :: (list_rem b es)
```

If $e = b$, then the result of the function is list_rem b es and also $a \in es$ because $a \neq b$ and $a \in$ ls. Therefore, by applying the inductive hypothesis, we have the thesis. In $e \neq b$, then the result of list_rem b ls is e :: (list_rem b es) then the thesis holds because, either a = e or else a $\in$ es and, by inductive hypothesis a $\in$ (list_rem b es) which gives the proof. □

list_diff l m   returns the list obtained by subtracting m from l and is defined as

```
let rec list_diff l = function
  | []      →  l
  | e::es →  list_diff (list_rem e l) es
```

Function list_diff enjoys the following property:

Lemma 9.2 *Let* l *and* m *be two lists. Then* $\forall$el $\in$ m.el $\notin$ (list_diff l m)*.*

Proof. We reason by induction on the length of m. If m = [] then the proposition trivially holds. Let m be e::es, then the result of the function is list_diff (list_rem e l) es. If el = e then, by Lemma 9.1, el $\notin$ (list_rem el l) and the thesis follows by the fact that the recursive call never adds anything to the result (avoiding the possibility of re-introducing el). On the other hand, if el $\neq$ e then el $\in$ es, and the inductive hypothesis concludes the proof. □

9.1 HD-automata states, labels and transitions

A generic automaton (see Definition 5.1) is made of four ingredients: States, initial state, labels and arrows. As far as finite state automata are concerned, it is possible to represent automata by enumerating states and transitions.

Observation 9.1 *We assume that* $1,..,n$ *are the names of a state having* n *names. This assumption does not imply any loss of generality because names are local to states. We reserve* 0 *for denoting name* $*$*, the symbol used for denoting name creation in transitions (see page 23). A symmetry over* n *names may be simply expressed by means of a list of distinct integers, each belonging to segment* $1,...,n$*; for instance if* ρ *is the permutation*

$$\begin{pmatrix} 1 & ... & n \\ i_1 & ... & i_n \end{pmatrix}$$

then the list $[i_1;...;i_n]$ *is a representation in terms of list of integers of* ρ*. For instance,* $[2;1;3]$ *represents a permutation of* 3 *elements: Namely, the permutation that exchanges* 1 *and* 2*, and leaves* 3 *unchanged.*

In this case we say that $[i_1;...;i_n]$ corresponds to ρ.

We adopt this conventions on names and permutations also for representing other functions on names. In particular, if $qd = \langle \ell, \pi, \sigma, q \rangle$ is a quadruple, π is represented by means of a list of integers `pi` whose length is $\mid \ell \mid$ and whose i-th position contains $\pi(i)$ (for $i = 1, ..., \mid \ell \mid$). Finally, σ is a list of integers `sigma` whose length is m, the number of names of q and whose i-th element is $\sigma(i)$ (for $i = 1, ..., m$). We say that `pi` (`sigma`) *corresponds to* π (σ).

As discussed in Section 1, HD-automata are an extension of ordinary automata in the sense that states and labels have a richer structure carrying information on names. A state may be described as a triple

```
type State_t =
   | State of id: string  *  names: int list  *  group: (int list) list
```

Where *id* is the name of the state; *names* are the local names of the state and are represented as a list of integers. The third component of a state is its *group* which contains those permutations that leave the state unchanged. By the previous observation, we can represent it as a list of list of integers.

Definition 9.1 (States correspondence) *An element* State(`q`, `names`, `group`) corresponds *to a state* q *of a named set* $A = \langle Q, \mid _ \mid, \leq, G \rangle$ *if, and only if,*

- $q \in Q$
- $\mid q \mid =$ `List.length names`
- `group` *corresponds to Group in terms of correspondence between lists and sets (i.e. reciprocal element-wise correspondence, as described in Section 9).*

We remark that, since, we concretely represents names as integers we exploit the integer order to induce an order to states; therefore, we do not explicitly mention it in Definition 9.1. Notice also that the first component of bundles is not represented. This is possible because a main design choice is that we always deals with bundles that are obtained by applying the iterative construction

$H_{i+1} = \widehat{K;\mathcal{T}(H_i)}$. Therefore, the first component of these bundles always is S_K, the set of states of the initial automaton.

Arrows are represented as triples with a *source* state, a *label* and a destination state.

```
type labeltype = string * int list * int list

type Arrow_t = Arrow of
  source: State_t * label: labeltype * destination: State_t
```

Type of arrows relies on type for labels which are triples whose first components are the name of the action (for π-agents, a string among $TAU, BOUT, OUT, BIN, IN$); the second component of a label is the list of names exposed in the transition; finally, the last component of a label is a function mapping names in the destination to names of the source state. An simpler alternative definition could have been obtained by embedding the `labeltype` in Arrow. Although more adherent to Definition 7.3, this solution is less general than the one adopted, because different transition systems have different labels.

Now we can give the structure which represents automata:

```
type Automaton_t =
  start: State_t *
  states: State_t list *
  arrows: Arrow_t list
```

The first component is the initial state of the transition system, then the list of *states* and *arrows* are given.

Bundles rely on quadruples over named sets (see Definition 7.3). Basically, a quadruple describes state transitions. Transitions are labeled and, our implementation represents part of information carried by quadruples into labels:

```
type quadtype = Qd of Arrow.labeltype * State_t

type Bundle_t = quadtype list
```

Type `quadtype` and Observation 9.1 allows us to state a precise connection between quadruples and objects that populate `quadtype`.

Definition 9.2 (Quadruple correspondence) *Given a quadruple* $qd = \langle \ell, \pi, \sigma, q \rangle$*, we say that* Qd((`lab`, `pi`, `sigma`), `q`) corresponds *to qd, if, and only if,*

- `lab` *is a string with value* ℓ*,*
- `pi` *corresponds to* π*,*
- `sigma` *corresponds to* σ*,*
- `q` *corresponds to* q*.*

Definition 9.3 (Automata correspondence) *Let* $K = \langle Q, \mathcal{T}(Q), k : Q \rightarrow \mathcal{T}(Q)\rangle$ *be a named function representing an automaton for a* π*-agent. We say that* $(\mathtt{q}, \mathtt{qs}, \mathtt{as})$ corresponds to K *iff,* `qs` *corresponds to* Q *and, for each* $qd \in k(q)$ *there exists* $\mathtt{a} \in \mathtt{as}$ *such that, if* $\mathtt{a} = (\mathtt{s}, (\mathtt{lab}, \mathtt{pi}, \mathtt{sigma}), \mathtt{t})$*, then* $\mathtt{Qd}((\mathtt{lab}, \mathtt{pi}, \mathtt{sigma}), \mathtt{t})$ *corresponds to* qd*.*

Previous definition allows us to easily compute $k(q)$ for each state q. Indeed, let us consider the function `Automaton.bundle` defined as

```
let bundle hda q =
  Bundle.from_arrow_list (List.filter
     (fun x -> 0 = State.compare (Arrow.source x) s) (arrows hda))
```

Proposition 9.1 *If* `hda` *and* `s` *are an HD-automata and a state which respectively correspond to* k *and* s*, then* `bundle hda s` *corresponds to* $k(s)$*.*

Proof. All arrows in `hda` are first filtered in order to select those of them whose source state is `s`, then function `Bundle.from_arrow_list` transforms each of them in the quadruple obtained by discharging the source from the arrow. □

Our representation of bundles, symmetries and function of names allows a simple representation of operation on bundles in terms of list manipulation. For instance, let us consider the function

$$\{\!|\, \beta \,|\!\} = \bigcup_{\langle \ell,\pi,\sigma,q\rangle \in Step_\beta} rng(\pi) \cup rng(\sigma) \setminus \{*\}$$

which yield the names of a bundle β and is implemented by function `bundle_names`, reported below.

```
let names = function Qd((lab,pi,sigma),target) -> (pi @ sigma)

let bundle_names bundle =
  Utils.unique (Utils.list_rem 0 (List.flatten (List.map names bundle)))
```

Function `bundle_names` applies `names` to each quadruple in the list `bundle` corresponding to β (`List.map names bundle`). This returns a list whose items are the names appearing in the quadruples of `bundle` that are obtained by merging the lists `pi` and `sigma`. Finally, all those lists of names are merged together (`List.flatten`), if present, 0 is removed (`Utils.list_rem`) and multiple occurrences are collapsed into a single occurrence (`Utils.unique`). It is easy to see that the following proposition hold:

Proposition 9.2 *If* `bundle` *corresponds to a bundle* β *then* $a \in \{\!|\, \beta \,|\!\}$ *if, and only if,* a *occurs in* `bundle_names bundle`*.*

Proof. By definition, $a \in \{\!|\, \beta \,|\!\}$ if, and only if, there exists a quadruple $qd = \langle \ell, pi, \sigma, q\rangle$ in $Step_\beta$ such that $a \in rng(\pi) \cup rng(\sigma) \setminus *$. Let `Qd((lab, pi, sigma), q)` be a quadruple in `bundle` which corresponds to β. Then, by definition `pi` =

$[\pi(1); ...; \pi(| \ \ell \ |)]$ and `sigma` $= [\sigma(1); ...; \sigma(m)]$, where $m = card(rng(\sigma))$ then $a \in$ `pi@sigma`. By observing that `pi@sigma` $\in$ (`List.map names bundle`). Then, since the flattening operation on lists corresponds to set union, we have that

$a \in$ `Utils.unique (Utils.list_rem 0 (List.flatten (List.map names bundle)))`.

Finally, by hypothesis, $a \neq *$ and, therefore, by Lemma 9.1. □

As stated in Section 7.1, normalization is the most important operation on bundles. It needs *red* function to be computed. Function *red* is implemented as follows:

```
let red bundle =
  let dominated =
    List.filter
      (fun qd → None <> (Domination.dominated qd bundle))
      bundle in
    list_diff bundle dominated
```

Proposition 9.3 *If* `bundle` *corresponds to a bundle* β *then* `red bundle` *corresponds to* $red(\beta)$.

Proof. First, let us observe that `dominated` is the list of quadruples which corresponds to the set of input quadruples that are redundant. Indeed, `bundle` is filtered according to the function `Domination.dominated`, that returns `None` only if `qd` is not redundant. Finally, by Lemma 9.2, `Utils.list_diff` removes from those transitions from `bundle`. □

```
let normalize red bundle =
  let w_bundle = red bundle in
  let an = bundle_names w_bundle in
    rename (list_diff bundle (List.filter (remove_in an) bundle))
```

Proposition 9.4 *If* `bundle` *corresponds to a bundle* β *then* `normalize red bundle` *corresponds to* $norm(\beta)$.

Proof. First, active names `an` are computed in order to filter `bundle` obtaining all redundant transition covered by some bound input transition. By Proposition 9.2 and Lemma 9.1 we can conclude that `an` corresponds to $\{\!|\ \beta\ |\!\}$. Lemma 9.2 and the fact that list filtering corresponds to test of set membership, ensure that from `bundle` all redundant transitions are removed. Indeed, `remove_in` is defined as

```
let remove_in an =
  function Qd((lab,pi,sigma),target) as qd →
    (lab = "in") && (not (List.mem (obj qd) an))
```

which computes exactly the redundancy condition. The last function applied to the so computed bundle is `rename` which shifts the local active names of a state with their position in the list of active names. Note that this is a safe operation because only the active names of a state are important and their meaning is local to the state, moreover, such renaming amount to compute the permutation of names that returns the normalized bundle as defined in Section 7.1. □

9.2 Block

The most important data structures are *blocks*. They represents action of the functor on states of the automata and contains all those information for computing the iteration steps of the algorithm expressed in a set theoretic framework. They represent both (finite) named functions and partitions of an automaton (at each iteration of the algorithm). Hence, at the last iteration a block corresponds to a state of the minimal automaton. A block has the following structure:

```
type Block_t =
  Block of
    id      : string  *
    states  : State_t list  *
    norm    : Bundle_t  *
    names   : int list  *
    group   : int list list  *
    Σ       : (State_t → (int  *  int) list list)  *
    Θ^-1    : (State_t → (int  *  int) list)
```

Field *id* is the name of the block and is used to identify the block in order to construct the minimal automaton at the end of the algorithm. Field *states* contains the states which are considered equivalent with respect the equivalence relation used in the algorithm[6]: In this case the early bisimulation relation. Remaining fields respectively represent

- the normalized bundle with respect to the block considered as state (*norm*),
- *names* is the list of names of the bundle in *norm*,
- *group* is its group,
- the functions relative to the bundle (Σ), last field, Θ^{-1}, is the function that, given a state q, maps the names appearing in *norm* into the name of q. Basically, $\Theta^{-1}(q)$ is the function which establishes a correspondence between the bundle of q and the bundle of the corresponding representative element in the equivalence class of the minimal automaton.

We draw (some components of) a block as in Figure 5: The upper elements are the states in the block, while the element `x` is the "representative state", namely it is a graphical representation of the block as a state. For each state q

[6] We recall that Mihda is parametrized with respect to the equivalence relation.

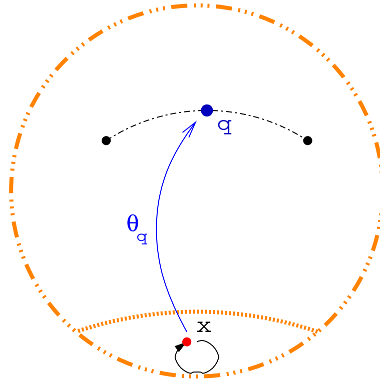


Figure 5: Graphical representation of a block

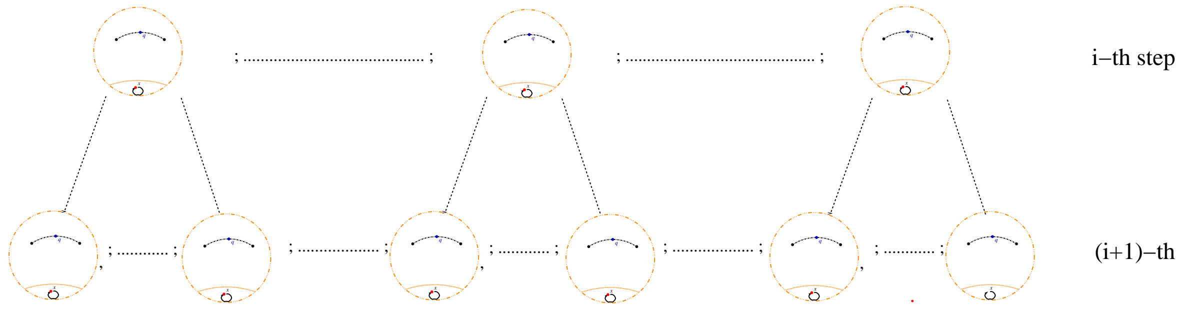


Figure 6: Graphical representation of an iteration step

a function $\theta_{\mathtt{q}}$ maps names of x into the names of q. Function $\theta_{\mathtt{q}}$ describes "how" the block approximates the state q at a given iteration. The circled arrow on x aims at recording that a block also has symmetries on its names. Bundle *norm* of block x is computed by exploiting the ordering relations over names, labels and states.

A graphical representation of an iteration step of Mihda is given in Figure 6. The idea is that each block in the list of current blocks is first splitted, as far as possible, into a number of *buckets*, i.e. quasi-blocks defined later. Then each bucket is transformed in a new block, namely, the lacking components are uniformly computed at the end of the splitting phase.

The main operation on a block is the operation which splits a block into buckets. A list of blocks is returned as result of each iteration. Such blocks represent the states of the current approximation of the minimal automaton. A bucket has the same fields of a block apart from the name, the group of symmetries and the functions mapping names of destination states in name of source states. Essentially, the split operation checks if two states in a block are equivalent or not. States which are no longer equivalent to the representative element of the block are removed and inserted into a bucket.

Given a bundle `bl`, a predicate over states `pred` and a block `block`, function `Block.split` returns a bucket and a block. The bucket collects the states of `block` which violate `pred`, while the returned block contains the remaining states.

```
let Block.split bl pred block =
  let eqv_chk = List.map (fun q → q, (pred q)) (states block) in
  let (eq_states, non_eq_states) =
    List.partition (fun (q,th) → th != None) eqv_chk in
  let new_states = (fst (List.split eq_states)) in
  let old_states = (fst (List.split non_eq_states)) in
  let new_inv_thetas = fun q →
    try
      invert ((function Some x→x) (List.assoc q eqv_chk))
    with Not_found → failwith "new_inv_thetas: exception" in
    (Bucket(new_states, bl, (Bundle.active_names bl), new_inv_thetas),
      (create
        (id block)
        old_states
        (norm block)
        (names block)
        (group block)
        (sigmas block)
        (inv_thetas block)))
```

It is worth to detail much more on the parameter `pred`. It is a function that, given a state `q`, returns an optional value that, roughly speaking, yields "the proof" for equivalence of `q` with respect to the other states of the bucket. More precisely, `pred` returns `None` if the equivalence does not hold, otherwise, a function θ mapping names of `q` into names of `bl` such that each arrow in the bundle of `q` appears in `bl` (and *viceversa*). Function θ is the function Θ^{-1} associates to `q` when the bucket is turned into a block.

Note that, the new block has the same component of the old one because at the end of the splitting phase, all the states of the initial block will be assigned to a bucket without considering the information contained in those fields.

Definition below states the correspondence between a list of blocks and a coalgebra.

Definition 9.4 (Block correspondence) *Let $K = \langle Q, \mathcal{T}(Q), h : Q \to \mathcal{T}(Q)\rangle$ be a transition system over named sets and π-actions. A list of blocks* `blocks` corresponds to K *when given $q, q' \in Q$ and their* ocaml *representation* `q` *and* `q'`*, then $h(q) = h(q')$ if, and only if*

- *there exists* `bl` $\in$ `blocks` *such that* `q` *and* `q'` *are in* `bl`

and

- `bl`.norm *corresponds to* $Step_{h(q)}$

10 The main cycle

Let us recall the iterative step introduced at the end of Section 7.2:

$$h_{H_{i+1}}(q) = norm\ \langle D_{H_i}, \{\langle \ell, \pi, \sigma'; \sigma, h_{H_i}(q')\rangle \mid q \xrightarrow{\ell,\pi,\sigma} q', \sigma' : \Sigma_{H_i}(q')\}\rangle. \quad (3)$$

For each state q of the automaton, $h_{H_{i+1}}(q)$ determines the normalized bundle associated with to q. Following equation (3), we can compute $h_{H_{i+1}}$ over a finite state automaton in the following steps:

a. determine the bundle of q in the automaton;

b. for each quadruple $\langle \ell, \pi, \sigma, q'\rangle$ in this bundle, apply h_{H_i} to q', the target state of the quadruple (yielding the bundle associated in the previous step to q');

c. left-compose this $\sigma' \in \Sigma(q')$ with σ;

d. normalize the resulting bundle.

This intuitive idea must be refined because Mihda represents h_{H_i} as a list of blocks. In this representation, $h_{H_i}(q)$ corresponds to field *norm*, namely the bundle of the block containing `q`, the state corresponding to q. A graphical representation of those steps in terms of blocks is depicted in Figure 7.

Step (a) is computed by the facility `Automaton.bundle` that filters all arrows of the automaton whose source corresponds to q. Figure 7(a) shows that a state `q` is taken from a block and its bundle is computed.
Step (b) is obtained by applying `Block.next` to the bundle of `q`. `Block.next` substitutes all target states of the quadruples with the corresponding current block and computes the new mappings as described in Figure 7(b).
Step (c) seems not correctly adhere to the corresponding step of equation 3, but if we consider that θ functions are computed at each step by composing σ's we can see that they exactly play the rôle of σ's.
Finally, step (d) is represented in Figure 7(d) and is obtained via the function `Bundle.normalize`. Observe that redundant transitions must be removed and other components of the new block must be computed as defined by Θ^{-1}.

In order to give an intuitive understanding of the split operation, we describe how states are separated. Let us assume an automaton and an equivalence relations over states (e.g. early bisimilarity) have been fixed. Let `pred` be a function such that, given a bundle `bundle` and a state `q`, returns `None` if the normalized bundle of `q` in the automaton is not equivalent to `bundle` according to the equivalence relation between states `pred`. Then we can divide the states of a given block with respect to `bundle` and `pred` into two different lists of states. More precisely, this operation relies on the `Block.split` function and returns a pair whose first component is a bucket and whose second component is a block. The bucket contains those states whose normalized bundle is not equivalent to `bundle`, while the block component contains the remaining states.

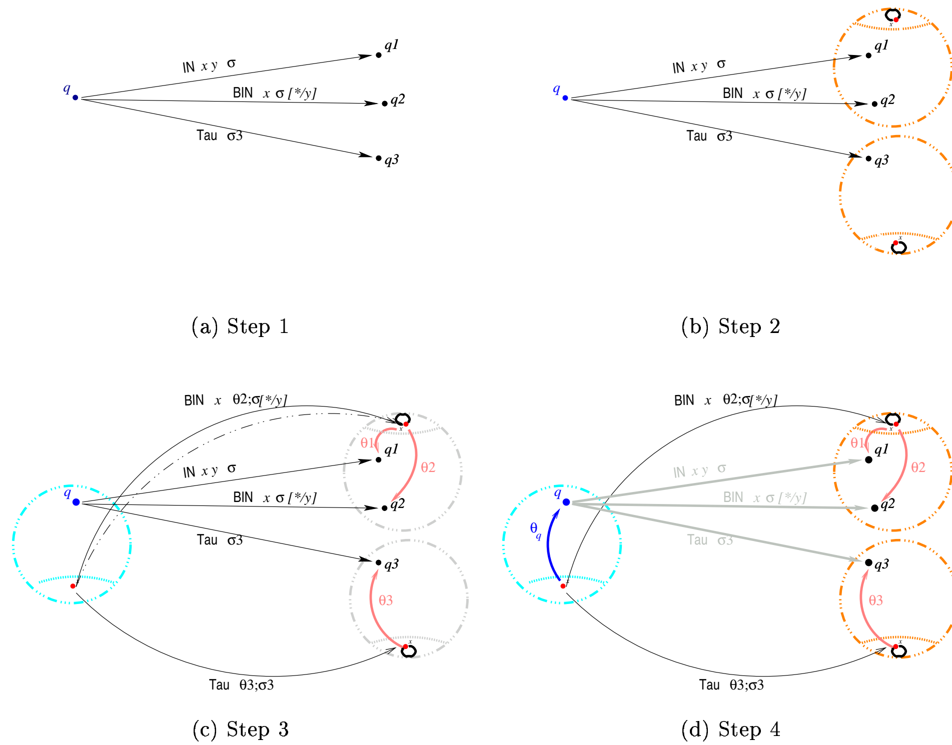


Figure 7: Computing $h_{H_{i+1}}$

The main part of Mihda consists of the cycle that computes the partitions of each iteration. Each block is splitted by iterating the application of function `split`.

```
let split blocks block =
  try
    let minimal =
      (Bundle.minimize red
        (Block.next
        (h_n blocks)
        (state_of blocks)
        (Automaton.bundle aut (List.hd (Block.states block))))) in
      Some (Block.split
        minimal
        (fun q →
          let normal =
            (Bundle.normalize
              red
              (Block.next (h_n blocks)
                (state_of blocks)
                (Automaton.bundle aut q))) in
            Bisimulation.bisimilar minimal normal)
        block)
  with Failure e → None
```

Let `block` be a block in the list `blocks`, function `split` computes `minimal` by minimizing the reduced bundle of the first state of `block`, and returns the optional value `Some(bk, block′)` if `Block.split` applied to `minimal`, to the bisimilarity relation and to `block` returns (`bk`, `block′`); otherwise `None` is returned.

Observation 10.1 *The choice of the state for computing* `minimal` *is not important: Without loss of generality, in fact, given two equivalent states* `q` *and* `q'`*, it is possible to map names of* `q` *into names of* `q'` *preserving their associated normalized bundle if, and only if, a similar map from names of* `q'` *into names of* `q` *exists.*

Moreover, we also point out that, minimization of a bundle with n names corresponds to normalize the bundle and replace each name with a name in $1, ..., n$ preserving the convention that names of a state are an initial segment of natural numbers.

Once `minimal` has been computed, `split` invokes `Block.split` with parameters `minimal`, `block`; the second argument of `Block.split` is a function that computes the (current) normalized bundle of each state in `block` and checks whether or not it is bisimilar to `minimal`.

This computation is performed by function `Bisimulation.bisimilar`. If bisimilarity holds through $\theta_{\mathtt{q}}$ then `Some` $\theta_{\mathtt{q}}$ is returned, otherwise `None` is returned.

We are now ready to comment on the main cycle of Mihda.

```
let blocks = ref [ (Block.from_states states) ] in
let stop = ref false in

  while not ( !stop ) do
    begin
      let oldblocks = !blocks in
      let buckets = split_iter (split oldblocks) oldblocks in
      begin
        blocks := (List.map (Block.close_block (h_n oldblocks)) buckets);
        stop :=
          (List.length !blocks) = (List.length oldblocks) &&
          (List.for_all2
            (fun x y → (Block.compare x y) == 0)
            !blocks
            oldblocks)
      end
    end
  done ;
  !blocks
```

Let `k` = (`start`, `states`, `arrows`) be a HD-automaton which corresponds to the coalgebra K. Initially, `blocks` is the list whose only item is a block containing all the states of `k`.

Observation 10.2 *By definition, initially,* `k` *corresponds to* H_0*, indeed, function* `Blocks.from_states` *puts all the states in the same block and assign the empty list to the field* norm *of such block.*

At each iteration, the list of blocks is splitted, as much as, possible by `split_iter` that returns the list of buckets. Then, by means of `Block.close_block`, all buckets are turned into blocks which are assigned to `blocks`. Finally, the termination condition `stop` is evaluated. Note that Theorem 7.1 states that D_{i+1} must be isomorphic to D_i. This condition is equivalent to say that a bijection can be established between `oldblocks` (that corresponds to D_i) and `blocks` (corresponding to D_{i+1}). Moreover, since order of states, names and bundles is always maintained along iterations, both lists of blocks are ordered. Hence, the condition reduces to test whether `blocks` and `oldblocks` have the same length and that blocks at corresponding positions are equal. More formally, the following theorem holds:

Theorem 10.1 *For each iteration i, at the end of the main cycle of* Mihda*,* `blocks` *corresponds to* h_{H_i}.

Proof. The proof is given by induction on the iteration step. The base of induction is trivial.

Let assume that, at the end of the i-th iteration, the theorem holds and, by contradiction, that, at the end of the $(i+1)$-th iteration `blocks` does not correspond to $h_{H_{i+1}}$. Then, by Definition 9.4, there are two states q and q' such that either

1. `q` and `q'` lies in different blocks

or else

2. both `q` and `q'` are in the same block `bl` but `bl`.*norm* does not corresponds to $Step_{h_{K_i}(q)}$.

Let us first consider Case (1). By hypothesis, $h_{K_{i+1}}(q) = h_{K_{i+1}}(q')$ then $h_{K_i}(q) = h_{K_i}(q')$ because it is not possible that states distinguished in a given iteration later become bisimilar. By inductive hypothesis, `blocks` at i-th iteration corresponds to h_{K_i}, hence `q` and `q'` are in the same block at the i-th iteration. States `q` and `q'` can be separated at the $(i+1)$-th iteration, if, by construction, `split` assigns them to different buckets. This can happen only if there exists a state whose minimized bundle computed in `minimal` can be put in correspondence by `Bisimulation.bisimilar` with the normalized bundle of `q` but not with the normalized bundle of `q'`. Since `minimize` simply renames bundles preserving the order of names, then, at the $(i+1)$-th iteration, the normalized bundle of `q` corresponds to $h_{K_{i+1}}(q) = h_{K_{i+1}}(q')$ but the normalized bundle of `q'` does not correspond to $h_{K_{i+1}}(q')$. This contradicts Proposition 9.4 that ensures that `normalize` correctly implements function *norm*.
Following the final part of the proof of Case (1), we can simply observe that Case (2) is not possible by construction. Indeed, the field `Block`.*norm* is built out from the normalized bundle of its states that, by Proposition 9.4, it must correspond to normalized bundles of the corresponding states of the automaton.
□

Theorems 10.1 and 7.1 also ensure that the termination condition is correctly computed because, by Theorem 7.1, a fix-point is reached and, at the terminal iteration `blocks` corresponds to the fix-point (Theorem 10.1). The successive iterations will not change any block therefore, the length of `blocks` and each block in it will not change. As stated above, `blocks` and its elements are maintained ordered along each iteration and therefore, checking whether `blocks` change or not can be executed, as expressed by the assignment to `stop` in the main cycle, checking if $\texttt{blocks}_i$ equals $\texttt{oldblocks}_i$ for any block $\texttt{blocks}_i \in$ `blocks`.

11 Concluding Remarks

The choice of implementing Mihda in ocaml has been driven by the functional and type-theoretic flavour of the co-algebraic specification. Both features fit well with ocaml programming characteristics. At a first glance, one can think that performance is undertaken: Surprisingly our benchmarks suggest that Mihda is not inefficient. For instance, we have considered the specification of the core of the handover protocol for the GSM Public Land Mobile Network proposed by the European Telecommunication Standards Institute [29]. The HD-automaton obtained by the π-calculus specification has 506 states and 745 transitions. The minimization of the handover protocol takes almost 9 seconds on an Athlon

1800+ under Linux RedHat 7.2, while 21 seconds are necessary on a Pentium III 500Mhz under Linux RedHat 7.1. The resulting HD-automaton consists of 105 states and 197 transitions.

The phase where Mihda spends the most part of computation time is the calculation of symmetries of names. For the time being Mihda trivially computes symmetries by generating the permutations and checking whether or not they change the behaviour of blocks. In this way the computational cost is factorial in the number of names. This implementation choice was suggested by the goal of producing a prototype of Mihda rapidly. However, the number of names remains very low in real cases. For instance, the specification of the GSM protocol initially consists of π-agents with 11 names. Instead, the average number of names per state in the compiled HD-automaton is 2.4: only two states have five names and more that three hundred states have only two names. We plan to enhance the efficiency of symmetries computation. Indeed, by considering that symmetries on new names added to a block does not affect already computed symmetries, we can compute new symmetries and "multiply" them with the old ones.

References

[1] Peter Aczel. Algebras and coalgebras. In Roy Backhouse, Roland Crole and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Constructio*, volume 2297 of *LNCS*, chapter 3, pages 79–88. Springer Verlag, April 2002. Revised Lectures of the Int. Summer School and Workshop.

[2] Peter Aczel and Nax Mendler. A final coalgebra theorem. In LNCS, editor, *Category Theory and Computer Science*, volume 389, 1989.

[3] Jirí Adámek, Stefan Milius, and Jirí Velebil. Final coalgebras and a solution theorem for arbitrary endofunctors. In Lawrence S. Moss, editor, *Coalgebraic Methods in Computer Science*, Grenoble, France, April 2002.

[4] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π–calculus. In U. Montanari and V. V. Sassone, editors, *Proc. of CONCUR'96*, volume 1119 of *LNCS*, pages 147–162. Springer, 1996.

[5] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[6] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. Observing localities. *Theoretical Computer Science*, 114(1):31–61, June 1993.

[7] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000. ISBN 2-84177-121-0.

[8] Mads Dam. Model Checking Mobile Processes. *Information and Computation*, 129(1):35–51, 1996.

[9] Jean Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[10] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Roberto Raggi, Gianluca Trentanni, and Emilio Tuosto. Verification on the web. Technical Report TR–02–18, Dipartimento di Informatica, Università di Pisa, Pisa (Italy), December 2002.

[11] GianLuigi Ferrari, Ugo Montanari, and Marco Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In Mogens Nielsen and Uffe Engberg, editors, *FOSSACS 2002*, volume LNCS 2303, pages 129–143. Springer Verlag, 2002.

[12] Gianluigi Ferrari, Ugo Montanari, and Paola Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1):53–103, November 1996.

[13] Cedric Fournet and George Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

[14] Cedric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile processes. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag.

[15] Ursula Goltz and Wolfgang Reisig. The non-sequential behaviour of petri nets. *Information and Computation*, 57(2/3):125–147, 1983.

[16] Mattew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. *[37]*, pages 95–115, 1999.

[17] Koei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–147, 1991.

[18] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1996.

[19] Bengt Jonsson and Joachim Parrow. Deciding bisimulation equivalences for a class of Non-Finite-State programs. *Information and Computation*, 107(2):272–302, December 1993.

[20] Paris C. Kanellakis and Scott A. Smolka. Ccs expressions, finite state processes and three problem of equivalence. *Information and Computation*, 86(1):272–302, 1990.

[21] Peter Lee, Fritz Henglein, and Neil D. Jones, editors. *Proc. of the ACM Symposium on Principles of Programming Languages.* ACM Press, 1997.

[22] Mihda is publically available at `http://jordie.di.unipi.it:8080/mihda`.

[23] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

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

[25] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[26] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[27] Ugo Montanari and Marco Pistore. History dependent automata. Technical report, Computer Science Department, Università di Pisa, 1998. TR-11-98.

[28] Ugo Montanari and Marco Pistore. pi-calculus, structured coalgebras, and minimal HD-automata. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 2000.

[29] Fredrik Orava and Joachim Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4(5):497–543, 1992.

[30] Catusha Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In ACM, editor, *[21]*, pages 256–265, New York, NY, USA, 1997. ACM Press.

[31] Joachim Parrow and Victor Bjorn. The fusion calculus: Expressiveness and symmetry in mobile processes. In *13th Annual IEEE Symposium on Logic and Computer Science.* IEEE Computer Society Press, 1998.

[32] Marco Pistore. *History dependent automata.* PhD thesis, Computer Science Department, Università di Pisa, 1999.

[33] Paola Quagli. *The π-calculus with explicit substitutions.* PhD thesis, Università di Pisa, Dipartimento di Informatica, 1996.

[34] James Riely and Mattew Hennessy. Trust and Partial Typing in Open Systems of Mobile Agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages (POPL'99)*, pages 93–104, 1999.

[35] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[36] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2002.

[37] Jan Vitek and Christian D. Jensen. *Secure Internet programming: security issues for mobile and distributed objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.

[38] David Walker. calculus semantics of object-oriented programming languages, 1995.

[39] James Worell. Terminal sequences for accessible endofunctors. In *Coalgebraic Methods in Computer Science*, number 19 in ENCTS, 1999.