Coalgebraic Minimisation of HD-Automata for the π-Calculus in a Polymorphic λ-Calculus*

Gianluigi Ferrari Ugo Montanari Emilio Tuosto

August 18, 2003

Abstract

We give a formal definition of HD-automata based on coalgebras. Exploiting such definition we formally specify and implement a minimisation algorithm for HD-automata derived from π-calculus agents. The algorithm is a generalisation of the partition refinement algorithm for classical automata and is specified as a coalgebraic construction defined using $\lambda^{\rightarrow,\Pi,\Sigma}$, a polymorphic λ-calculus with dependent types. The convergence of the algorithm is proved; moreover, the correspondence of the specification and the implementation is shown.

1 Introduction

Finite state automata (e.g. labelled transition systems) provide a foundational model underlying effective verification techniques of concurrent and distributed systems. From a theoretical point of view, many behavioural properties of concurrent and distributed systems can be naturally defined directly as properties over automata. From a practical point of view, efficient algorithms and verification techniques have been developed and widely applied to case studies of substantial complexity in several areas of computing such as hardware, compilers, and communication protocols. We refer to [2] for a review.

A fundamental property of automata is the possibility, given an automaton, to construct its canonical form: The minimal automaton. The theoretical foundations guarantee that the minimal automaton is indistinguishable from the original one with respect to many behavioural properties (e.g., bisimilarity of automata) and properties expressed in suitable modal or temporal logics. Minimal automata are very important also in practice. For instance, the problem of deciding bisimilarity is reduced to the problem of computing the minimal transition system [17, 3, 8]. The algorithm yields the minimal realization of the initial automaton by "grouping" all the equivalent states in a single class. Moreover, it is often convenient, from a computational point of view, to verify

---

*This work has been supported by EU-FET project **PROFUNDIS** IST-2001-33100 and by Italian MIUR projects **NAPOLI** and **COMETA**.

properties on the minimal automaton rather than on the original one. Indeed, minimisation algorithms can be used to attack the state explosion: They yield a small state space, but still retain all the relevant information for the verification.

One of the main advantages of applying formal methods to system design and specification is the possibility of constructing an abstraction of systems and their computations that are, at least at a certain extent, amenable of automatic verification. Many techniques have been studied, implemented and used for verifying systems. For instance, it is possible to verify whether an implementation "adheres" to its specification by checking a suitable behavioural equivalence among them. Another example is the *information leak* detection; for instance, in [6] the analysis of information flow is done by modelling the system as a CCS-process P and then verifying that it is equivalent to $\mathit{restr}(P)$, another process obtained by opportunely restricting the behaviour of P. A similar idea has been exploited in [1] for analysing cryptographic protocols in the spi-calculus.

Global computing systems consist of networks of stationary and mobile components. The primary features of a global computing systems are that components are autonomous, software versioning is highly dynamic, the network coverage is variable and often components reside over the nodes of the network (WEB services), membership is dynamic and often ad hoc without a centralised authority. Global computing systems must be made very robust since they are intended to operate in potentially hostile environments. Moreover, they are hard to construct correctly and very difficult to test in a controlled way. Although significant progresses have been made in developing foundational models and effective techniques to support formal verification of global computing systems, current software engineering technologies provide limited solutions to some of the issues discussed above.

Name passing calculi (e.g. the π-calculus [10, 9, 23]) are basically the best known and acknowledged models which provide a rich set of techniques for reasoning about mobile systems. *History Dependent automata* (HD-automata shortly) have been proposed in [18, 14, 15, 4] as a new effective model for name passing calculi. Similarly to ordinary automata, HD-automata are made out of states and labelled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation or name extrusion: These are the basic linguistic mechanisms of name passing calculi.

It is important to emphasise that names of states of HD-automata have *local meaning*. For instance, assume that $A(x, y, z)$ denotes an agent having three (free) names x, y and z. Then agent $A(y, x, z)$ obtained through the transformation which swaps names x and y is syntactically different from $A(x, y, z)$. However, these two agents can be "semantically" represented by means of a single state q in a HD-automaton simply by considering a "swapping" operation on the local names (corresponding to) names x and y.

Depending on the level of abstraction, different definitions of HD-automata can be given. On the one hand, HD-automata can be though of as the natural model for name passing calculi. Indeed, they have been characterised as automata over a permutation algebra, whose ingredients are sets of names and groups of permutations (renaming substitutions) on them. This framework is sufficient to describe and reason about formalisms with name-binding operations. It includes various kinds of transition systems providing syntax-free models of name-passing calculi [7, 15, 19]. On the other hand, at a more concrete level, HD-automata have been defined on the notions of *named sets* and *named functions*. Set elements are equipped with names which are defined up to specific groups of name permutations called *symmetries*. HD-automata are defined as coalgebras on a category whose objects are named sets and whose arrows are named functions. The two definitions are equivalent [15], with orbits of states that differ only for renaming of their local names in the first approach corresponding to a single state in the second approach. The two approaches are complementary, since the former is more natural, but yields infinite automata in all but the simplest cases, while the latter generates finite (actually quite compact) automata in many useful cases. General results concerning coalgebras guarantees the existence of the minimal HD-automaton up to bisimilarity. In [4] two of the authors specify a declarative coalgebraic procedure to perform minimisation of (finite state) HD-automata according to the second definition. The algorithm is a generalisation of the partition refinement algorithm for minimising ordinary automata (up to bisimilarity) [8].

This paper illustrates the development of a toolkit, called Mihda, providing facilities to minimise labelled transition systems for name passing calculi. Taking advantage of the coalgebraic specification of HD-automata and of the minimisation algorithm specified in [4], we formally define the general scheme for minimising labelled transition systems induced by name passing calculi. We introduce the concept of *kernel* of a named function that is very useful in the formal specification of such algorithm. Certain named functions[1] correspond to iterations of the algorithm and their kernels give the partitions of the states that correspond to current refinements of the minimal automaton. Moreover, kernels also have a very neat counterpart in Mihda; in particular, the main data structure of Mihda are *blocks* that represent the equivalence classes induced by the kernel of the current iteration.

A distinguished feature of this paper is the exploitation of a polymorphic λ-calculus, $\lambda^{\to,\Pi,\Sigma}$ [13], for describing data and control components of the minimisation procedure. The type system of $\lambda^{\to,\Pi,\Sigma}$ encompasses polymorphic and dependent types. We exploit polymorphism for abstracting from unimportant features (with respect to the minimisation algorithm); for example, it does not matter which is the type used for representing the states of HD-automata, the relevant information being the number of names and the symmetry of each state. Dependent types are useful for expressing functional dependencies among the components of a given construction. For instance, the type of the symmetries of a named set element includes those groups of permutations which act on the names of the element. The calculus $\lambda^{\to,\Pi,\Sigma}$ is also an effective basis

- to drive the implementation choices; for instance, the specification natu-

---

[1] These named functions approximate the unique arrow to the final coalgebra.

rally suggests an ML-like language (we chose ocaml) since the type system of $\lambda^{\to,\Pi,\Sigma}$ is a generalisation of the ML type system. Also, the generic modules of Mihda should be as general as possible with respect to both the actions labelling the transitions and the notion of bisimilarity actually employed;

- to show the correctness of the implementation.

In general, the final coalgebra is characterised as the maximal fixpoint of the functor and requires a recursive type structure. However, in the finite case, the minimal automaton is obtained in a finite number of iterations. Thus, interestingly enough, in our case recursive types are not needed to perform minimisation. The reason is that polymorphism allows us to pass to the functor a different type at each iteration.

Structure of the paper Section 2 collects the formal ingredients our work is based on. More precisely, Section 2.1 introduces the basic definitions of coalgebras and shortly discusses their adequacy for representing transition systems. Section 2.2 describes $\lambda^{\to,\Pi,\Sigma}$. Finally, Section 2.3 briefly reviews π-calculus and its early semantics.

The main results of the paper are in Section 3. In Section 3.1 the types for the coalgebraic presentation of HD-automata are given. Section 3.2 introduces the formal coalgebraic specification of HD-automata for the π-calculus. Section 3.2.1 details the types needed by π-calculus coalgebras. Section 3.2.2 specifies in $\lambda^{\to,\Pi,\Sigma}$ some auxiliary operations exploited in the definition of the functor, that is presented in Section 3.2.3. Section 3.2.4 defines how π-agents can be mapped into HD-automata (preserving early bisimulation). The minimisation algorithm is given in Section 3.3 where also its convergence on finite HD-automata is proved.

Section 4 shows the correspondence between the $\lambda^{\to,\Pi,\Sigma}$ specification of the minimisation algorithm and Mihda, its concrete realisation in ocaml.

Section 5 contains some concluding remarks.

2 Preliminaries

This section collects the three ingredients used in the rest of the paper; namely, coalgebras, $\lambda^{\to,\Pi,\Sigma}$ and the π-calculus.

2.1 Coalgebras

Coalgebras provides a very elegant mathematical machinery to describe the behaviour of process calculi. More generally, coalgebras can be used as an expressive formalism for defining transition systems (e.g. automata).

This section reviews some elementary notions of coalgebras. In particular, we will restrict our attention on coalgebras over sets and functions.

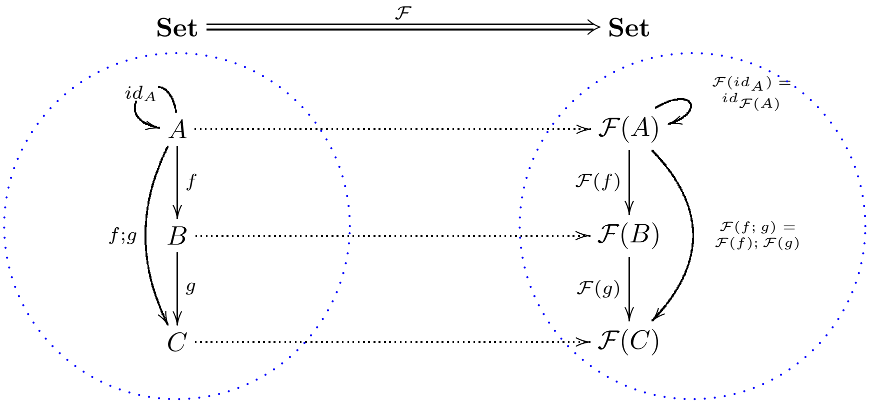


Figure 1: Functor over **Set**

Definition 2.1 (Functor over Set) *An* (endo-)functor $\mathcal{F}$ *over* **Set** *maps sets to sets and functions to functions as follows:*

- *for each function* $f : A \to B$, $\mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(B)$*;*
- *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 1 gives a graphical representation of how a functor acts over the category of sets and functions.

The identity mapping of sets and functions, or the mapping that associates a constant set L to any set A are functors overs **Set**. Another functor that will be very important in defining coalgebras is the *powerset functor*. 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 $\wp(f;g) = \wp(f);\wp(g)$.

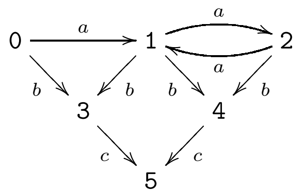
This proves that the powerset operation is functorial.

Definition 2.2 (Coalgebras, cohomomorphism) *Let $\mathcal{F}$ be an endo-functor on the category* **Set**. *A $\mathcal{F}$-*coalgebra *consists of a pair (A, α) such that $\alpha : A \to \mathcal{F}(A)$.*

*Let $(A, \alpha), (B, \beta)$ be $\mathcal{F}$-coalgebras. A function $f : A \longrightarrow B$ is called a $\mathcal{F}$-*cohomomorphism *if $\alpha; \mathcal{F}(f) = f; \beta$.*

A $\mathcal{F}$-coalgebra is a pair $\langle A, \alpha : A \to \mathcal{F}(A) \rangle$ where *alpha* is a function that, 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 $\langle Q, \alpha : Q \to L \times Q \rangle$ can be though of as a deterministic 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 labelled l. Similarly, let $\mathcal{TS}(X) = \wp(L \times X)$, then the coalgebra $\langle Q, \alpha : Q \to \wp(L \times Q) \rangle$ defines a labelled transition system over L.

Example 2.1 *Let us consider a finite-state automaton and its coalgebraic formulation via the mapping α.*



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

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

A $\mathcal{F}$-coalgebra (A, α) is *final* provided that for any A $\mathcal{F}$-coalgebra (B, β) there exists precisely one cohomomorphism $f : (B, \beta) \to (A, \alpha)$. Final coalgebras enjoy some interesting properties: If (A, α) is a final coalgebra then α is an isomorphism and A can be regarded as giving the canonical solution of the equation $A = \mathcal{F}(A)$.

Final coalgebras do not always exist. For instance, standard cardinality arguments can be exploited to show that the powerset functor $\wp(\_)$ does not admit final coalgebra. For many functors over sets, however, the final coalgebra exists. It is well know that for continuous functors the final coalgebra is obtained as the limit of the following sequence

$$\mathbf{1} \xleftarrow{!} \mathcal{F}(\mathbf{1}) \xleftarrow{\mathcal{F}(!)} \mathcal{F}^2(\mathbf{1}) \xleftarrow{\mathcal{F}^2(!)} \ldots$$

where $! : \mathcal{F}(\mathbf{1}) \to \mathbf{1}$ is the unique morphism from $\mathcal{F}(\mathbf{1})$ to the one element set $\mathbf{1}$. For instance, *polynomial* functors are continuous and hence have a final systems [20].

The class of polynomial functors consists of all the functors that we can build from the constant functor, the identity functor, sum, product and the function space functor. Notice that the powerset functor is not polynomial. However, the functor

$$\wp_{\mathrm{fin}}(S) = \{S' \mid S' \subseteq S \text{ and } S' \text{ finite}\}$$

has a final coalgebra. The powerset functor $\wp_{\mathrm{fin}}(\_)$ is the standard example of *bounded* functor. It has been proved that bounded functors have a final systems [20]. These results on the existence of the final systems have been extended to general categorical structures [26].

2.2 Overview of $\lambda^{\rightarrow,\Pi,\Sigma}$

This section reviews some basic type-theoretic notions underlying the description of languages which support the organisation of applications into autonomous (compilable) modules exploiting explicit type information (e.g. the ML module language). We refer to [13] for a detailed introduction to these issues.

In type theory, a polymorphic type describes a structure "having many types". Two powerful constructs to describe polymorphic types are the *general product and sum* types. A function type $t \rightarrow t'$ describes the type of a function mapping elements of t into elements of t'. Sometimes to specify the dependence of the result type on the value of the argument type (i.e. the type t' is an expression with free variable x of type t), the function type is written as $\prod_{x:t} t'$. This function type is called *general product* of t' over the index set t. The elements of this type are functions f such that $f(a) : t'[a/x]$, for each $a : t$.

A type $\langle t, t' \rangle$ describes a pair whose components are elements of type t and elements of type t'. When the value of the type of the first component determines the value of the type of the second component (i.e. the type t' is an expression with free variable x of type t) the pair type is written as $\sum\limits_{x:t} t'$. This type is called *general sum* of t' over the index set t. The elements of this types are pairs $\langle a, b \rangle$ with $a : t$ and $b : t'[a/x]$. General sums are equipped with projections to extract their components.

Generic sum types encompass tuple types; indeed, $\sum\limits_{x:t} t'$ is equivalent to $t \times t'$, if x does not occur free in t'. In these case, we will sometime write $t \times t'$ instead of $\sum\limits_{x:t} t'$.

We now introduce $\lambda^{\rightarrow,\Pi,\Sigma}$ the predicative calculus with products and sums. We let U_1 and U_2 to denote the universes of *non*-polymorphic (basic) and polymorphic types, respectively. We assume that the universe of non-polymorphic types contains a collection of *type constructors* and it is closed under product and function space (notice that U_1 does not belong to U_2). This allows us to assume type constructors such as lists, trees or enumeration types over a type t as basic structures of our calculus. The universe of polymorphic types can be made as rich as the universe of basic types.

The syntax of (pre-)terms of $\lambda^{\rightarrow,\Pi,\Sigma}$ is given by:

$$\begin{array}{lcl} M & ::= & U_1 \mid U_2 \mid bt \mid M \rightarrow M \mid \prod_{x:M} M \mid \sum_{x:M} M \\ & \mid & x \mid c \mid \lambda x : M.m \mid MM \\ & \mid & \langle x : M = M, M : M \rangle \mid \mathrm{I}(M) \mid \mathrm{II}(M) \end{array}$$

The first line of the grammar gives the syntax of the *type expressions*, the third line describes the structure associated with general sums; expressions $\mathrm{I}(M)$ and $\mathrm{II}(M)$ are the projections on the components of a pair (i.e. $\mathrm{I}(\langle M_1, M_2 \rangle) = M_1$ and $\mathrm{II}(\langle M_1, M_2 \rangle) = M_2$). The second line gives the productions of a (typed) λ-calculus. As usual, we will use τ to denote a term whenever the term is intended to be a type.

The type system is defined by type judgements $\Gamma \rhd M : \tau$ where Γ is type context of the form $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, giving the types of variables $x_1, \ldots, x_n$. Any of the types τ_i can be a basic type or a polymorphic type. Type contexts have to satisfy some well-formedness constraints. Here, we do not present all the inference rules which express when a context is well formed. To give the flavour of the type system we present, instead, a sample of the typing rules. In particular, we will focus on general products and sums.

The following inference rules describes the conditions for forming and handling general products.

$$\frac{\Gamma \rhd \tau : U_2 \qquad \Gamma, x : \tau \rhd \tau' : U_2}{\Gamma \rhd \prod_{x:\tau} \tau' : U_2} \qquad \frac{\Gamma \rhd M : \prod_{x:\tau} \tau' \qquad \Gamma \rhd N : \tau}{\Gamma \rhd MN : \tau'[N/x]}$$

$$\frac{\Gamma \rhd \tau : U_2 \qquad \Gamma, x : \tau \rhd \tau' : U_2 \qquad \Gamma, x : \tau \rhd M : \tau'}{\Gamma \rhd \lambda x : \tau.M \ : \ \prod_{x:\tau} \tau'}$$

The four inference rules below provide the conditions for forming general sums and for handling the associated terms.

$$\frac{\Gamma \rhd \tau : U_2 \qquad \Gamma, x : \tau \rhd \tau' : U_2}{\Gamma \rhd \sum_{x:\tau} \tau' : U_2}$$

$$\frac{\Gamma \rhd M : \sum_{x:\tau} \tau'}{\Gamma \rhd \mathrm{I}(M) : \tau} \qquad \frac{\Gamma \rhd M : \sum_{x:\tau} \tau'}{\Gamma \rhd \mathrm{II}(M) : \tau'[\mathrm{I}(M)/x]}$$

$$\frac{\Gamma \rhd M : \tau \qquad \Gamma, x : \tau \rhd \tau' : U_2 \qquad \Gamma \rhd N[M/x] : \tau'[M/x]}{\Gamma \rhd \langle x : \tau = M, N : \tau' \rangle : \sum_{x:\tau} \tau'}$$

2.3 The π-calculus

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

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

$$\alpha ::= \tau \;\big|\; \bar{x}y \;\big|\; xy,$$

where $r(A)$ is the rank of the *process identifier* A. The occurrences of y in $x(y).P$ and $\nu\, yP$ are bound; *free names* are defined as usual and $\mathrm{fn}(P)$ indicates the set of free names of agent P. For each identifier A there is a definition $A(y_1, \ldots, y_{r(A)}) := P_A$ (with y_i all distinct and $\mathrm{fn}(P_A) \subseteq \{y_1 \ldots y_{r(A)}\}$) and we assume that each identifier in P_A is in the scope of a prefix (guarded recursion).

The *observable actions* that agents can perform are defined by the following syntax:

$$\mu ::= \tau \;\big|\; \bar{x}y \;\big|\; \bar{x}(z) \;\big|\; xy;$$

where x and y are free names of μ ($\mathrm{fn}(\mu)$), whereas z is a bound name ($\mathrm{bn}(\mu)$); finally $\mathrm{n}(\mu) = \mathrm{fn}(\mu) \cup \mathrm{bn}(\mu)$. Usually, x is the *subject* name whereas y and z are called *object* names.

The SOS rules for the *early operational semantics* are defined in Table 1. As usual, we consider π-agents up-to *structural equivalence* $\equiv$ defined as the smallest congruence with respect to

- the monoidal laws for the parallel anch choice operators,
- α-conversion of bound names,
- $[x = y]\mathbf{0} \equiv \mathbf{0}$,
- $(\nu\, x)(\nu\, y)P = (\nu\, y)(\nu\, x)P$ and
- $(\nu\, x)(P \mid Q) \equiv P \mid (\nu\, x)Q$.

Several bisimulation equivalences have been introduced for the π-calculus [23]; they are based on direct comparison of the observable actions π-agents can perform. They can be strong or weak, early, late [10] or open [22]. In this paper we consider early bisimilarity since it provides the simplest setting for presenting the basic results of our framework. However, it is possible to treat also other behavioural equivalences and other dialects of the π-calculus (e.g. asynchronous π-calculus) [18].

Definition 2.3 (Early bisimulation) *A binary relation $\mathcal{B}$ over a set of agents is a strong early bisimulation if it is symmetric and, whenever $P \;\mathcal{B}\; Q$, we have that:*

- *if $P \xrightarrow{\mu} P'$ and $\mathrm{fn}(P, Q) \cap \mathrm{bn}(\mu) = \emptyset$, then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \;\mathcal{B}\; Q'$.*

Two agents are said strong early bisimilar*, written $P \simeq Q$, if there exists a bisimulation $\mathcal{B}$ such that $P \;\mathcal{B}\; Q$.*

$$\text{TAU } \tau.P \xrightarrow{\tau} P \qquad \text{OUT } \bar{x}y.P \xrightarrow{\bar{x}y} P \qquad \text{IN } xy.P \xrightarrow{xz} P\{z/y\}$$

$$\text{SUM } \frac{P_1 \xrightarrow{\mu} P'}{P_1 + P_2 \xrightarrow{\mu} P'} \qquad \text{PAR } \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2} \text{ if } \mathrm{bn}(\mu) \cap \mathrm{fn}(P_2)) = \emptyset$$

$$\text{COM } \frac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'} \qquad \text{CLOSE } \frac{P_1 \xrightarrow{x(y)} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} \nu\, y(P_1' \mid P_2')} \text{ if } y \notin \mathrm{fn}(P_2)$$

$$\text{RES } \frac{P \xrightarrow{\mu} P'}{\nu\, xP \xrightarrow{\mu} \nu\, xP'} \text{ if } x \notin \mathrm{n}(\mu) \qquad \text{OPEN } \frac{P \xrightarrow{\bar{x}y} P'}{\nu\, yP \xrightarrow{x(z)} P'\{z/y\}} \text{ if } x \neq y, z \notin \mathrm{fn}(\nu\, yP')$$

$$\text{MATCH } \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'} \qquad \text{IDE } \frac{P_A\{y_1/x_1, \ldots, y_{r(A)}/x_{r(A)}\} \xrightarrow{\mu} P'}{A(y_1, \ldots, y_{r(A)}) \xrightarrow{\mu} P'}$$

$$\text{STRUCT } \frac{P \equiv P' \xrightarrow{\mu} Q' \equiv Q}{P \xrightarrow{\mu} Q}$$

Table 1: Early operational semantics.

3 A minimisation procedure for HD-Automata

This section introduces the formal definitions for types and operations exploited to define the minimisation algorithm on HD-automata for π-agents. We take advantage of definitions given in [4] where an informal "type-based" specification of the minimisation algorithm has been given. Basically, our approach consists of formally describing the data and the control components of the minimisation procedure as $\lambda^{\rightarrow,\Pi,\Sigma}$ expressions. This provide us some benefits. First, it enables us to formally prove termination of the minimisation algorithm. Second, the specification given here is a refinement of [4] and it can be considered as an intermediate step toward the effective implementation, Mihda. In Section 4 we will show the strict correspondence between the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification and Mihda.

Remark 3.1 *In order to avoid cumbersome details regarding the definitions of $\lambda^{\rightarrow,\Pi,\Sigma}$ functions, we make some simplifying assumptions. First, we assume as given the primitive types (e.g. boolean, integers, strings...) and also that the type expressions include enumeration types. Second, we consider sets and operations on sets as primitive in our type language. This is not problematic since we will deal with finite collection of elements.*

3.1 Types for HD-automata

A first choice concerns the representation of names. Names must be totally ordered because names have a meaning local to the state of HD-automata, hence, they can be arbitrarily renamed.

Instead of considering *abstract* names (as done in the presentation of HD-automata [15]) we exploit a concrete representation of names in terms of natural numbers ω (with the usual order). We also need to represent finite sets of names, hence we let N to be the type defined as

$$\mathtt{N} \triangleq \prod_{n:\omega} 1 \cdots n$$

It is useful to reserve integer 0 for a special purpose, i.e. it always denotes a newly generated name. Hence, 0 only appears in transition labels, while names local to states start from 1. Moreover, we will consider functions from totally ordered sets to ω, hence we introduce the following definition:

Definition 3.1 ($\trianglelefteq$) *Given two totally ordered sets* $(A_1, \lesssim_1)$ *and* $(A_2, \lesssim_2)$*, we let* $\trianglelefteq$ *be the relation on functions from* A_1 *to* A_2 *such that* $f \trianglelefteq g$ *if, and only if,*

- *either,* $\forall q \in A_1.f(q) = g(q)$
- *or, there is* $q \in A_1$ *such that* $f(q) \neq g(q)$ *and* $\forall q' \in A_1.q' \lesssim_1 q \implies f(q') = g(q') \wedge f(q) \lesssim_2 g(q)$.

Let us remark that $\trianglelefteq$ is an order relation on functions from A_1 to A_2 and it is a computable relation if A_1 is finite.

Named sets represent states of HD-automata. The type of named sets is

$$\mathtt{NS} \triangleq \sum_{Q:\mathtt{U}_2} \sum_{\lesssim:Q\times Q\to\mathtt{bool}} \sum_{|\_|:Q\to\omega} \prod_{q:Q} \wp_{\mathrm{fin}}(\mathtt{N}(|q|) \to \mathtt{N}(|q|)).$$

As a matter of notation, for denoting the component of a named set A, we write Q_A, $\lesssim_A$, $|\_|_A$ and $\mathcal{G}_A$ in place of using the unwieldy (and less readable) notation of $\lambda^{\to,\Pi,\Sigma}$ based on projections I(_) and II(_). Given a named set A, we write $a \in A$ instead of $a : Q_A$.

A named set A lives in a generalised sum type[2] whose first component is a type Q_A, the second component is again a sum type with a function $\lesssim_A$ that represents a total order on Q_A and will be used for determining the canonical representative in a set of of states; function $|\_|_A$ is called the weight function (of A) and associates the number of names to elements in Q_A; the generalised product type of the last component assigns a set of permutations of names of q. In the following we write $q \lesssim_A q'$ ($q \not\lesssim_A q'$) instead of writing $\lesssim_A (q, q') = \mathtt{true}$ ($\lesssim_A (q, q') = \mathtt{false}$).

Transitions among states are represented by means of *named functions*:

Definition 3.2 (Named functions) *The type of* named functions *is defined as follows:*

$$\mathtt{NF} \triangleq \sum_{S:\mathtt{NS}} \sum_{D:\mathtt{NS}} \sum_{h:Q_S\to Q_D} \prod_{q:Q_S} \wp_{\mathit{fin}}(\mathtt{N}(|h(q)|_D) \to \mathtt{N}(|q|_S))$$

---

[2] Formally, A should be written as $\langle Q_A, \langle \lesssim_A, \langle |\_|_A, \mathcal{G}_A \rangle\rangle\rangle$. However, when no confusion arises, we prefer the more compact notation $\langle Q_A, \lesssim_A, |\_|_A, \mathcal{G}_A\rangle$ that avoids writing many brackets. The same notational abuse is adopted for the other sum types that will later appear in the paper.

It is worth fixing some notations that will help reading. Given a named function $H : \mathtt{NF}$*, we use the following abbreviations:*

- $\mathrm{dom}_H = \mathrm{I}(H)$,
- $\mathrm{cod}_H = \mathrm{I}(\mathrm{II}(H))$,
- $\mathrm{h}_H = \mathrm{I}(\mathrm{II}(\mathrm{II}(H)))$,
- $\Sigma_H = \mathrm{II}(\mathrm{II}(\mathrm{II}(H)))$,

which correspond to the projections of the sum type `NF`.

We implicitly assume that, for all elements $q \in \mathrm{dom}_H$

1. $\forall \sigma \in \Sigma_H(q).\mathcal{G}_{\mathrm{cod}_H}(\mathrm{h}_H(q)); \sigma = \Sigma_H(q)$,
2. $\forall \sigma \in \Sigma_H(q).\sigma; \mathcal{G}_{\mathrm{dom}_H}(q) \subseteq \Sigma_H(q)$,
3. *any function of* $\Sigma_H(q)$ *is injective.*

The type of named functions is a generalised sum type containing the named sets for source and destination (notice that the type of the destination does not depend on the type of the source), a mapping h from the source to the destination and, for each q in the source, there is a set of functions from the names of $h(q)$ to names of q.

The intuition behind conditions *1*, *2* and *3* is clear if we interpret a named function H as a coalgebraic description of a HD-automaton: Elements in $\mathrm{h}_H(q)$ are the possible transitions out of q and $\Sigma_H(q)$ are the mappings of names of target states of those transitions to names of q. Symmetry of $\mathrm{h}_H(q)$ are those permutations of names that when applied do not change $\mathrm{h}_H(q)$, the set of transition from q. Condition *1* states that any permutation is in the symmetry of $\mathrm{h}_H(q)$ if, and only if, when it is applied to any name correspondence σ from the names of the transitions to the names of q yields a map in $\Sigma_H(q)$ (which corresponds to the intuition of $\mathcal{G}_{\mathrm{cod}_H}(\mathrm{h}_H(q))$). Condition *2* states that the group of the starting state q does not *generate* transitions that are not in $\Sigma_H(q)$. Finally, condition *3* ensures that any name of the target state has a unique "meaning" along transitions from q.

An ordinary function f on sets induces a natural partition on its domain. Indeed, the relation $=_f \subseteq \mathrm{dom}_f \times \mathrm{dom}_f$ defined by $e =_f e' \iff f(e) = f(e')$ is an equivalence relation. The *kernel of* f ($\ker f$) is the partition induced on dom_f by $=_f$. Let f and g be two functions such that $\mathrm{dom}_f = \mathrm{dom}_g$, we can define $f \simeq g \iff \ker f = \ker g$, which trivially is an equivalence relation on functions with common domain. We can lift the concept of kernel to named functions.

Definition 3.3 (Kernel of named functions) *The* kernel of named function *H is the named set such that:*

- *The underlying set is* $\ker \mathrm{h}_H$*;*
- *the order relation, taken two classes* $A, B \in \ker \mathrm{h}_H$*, yields* `true` *if, and only if,* $\mathrm{h}_H(a) \lesssim_{\mathrm{cod}_H} \mathrm{h}_H(b)$*, for* $a \in A$ *and* $b \in B$*;*
- *the weight of a class* $A \in \ker \mathrm{h}_H$ *is* $|\mathrm{h}_H(a)|_{\mathrm{cod}_H}$*, for* $a \in A$*;*

- *the group of* $A \in \ker \mathrm{h}_H$ *is* $\mathcal{G}_{\mathrm{cod}_H}(\mathrm{h}_H(a))$, *for* $a \in A$.

Intuitively, $\ker H$ is obtained by considering the partition induced by h_H on its domain and by exploiting the named set structure of cod_H for the order, weight and group components. Notice that, in Definition 3.3, those components do not depend on the choice of a or b, since any element in $Q_{\ker H}$ is a set whose elements have the same image through h_H.

We already emphasised that named functions are used for defining a generic step of the iterative minimisation algorithm. Intuitively, named functions map states of the automaton in a minimal representative state (at the current iteration). In Section 4 the notion of kernel of a named function is exploited for specifying *blocks*, the main data structure of Mihda. Basically, a block collects those states considered equivalent at a generic iteration. Hence, intuitively the block *corresponds* to an element of the partition induced by the kernel of the named function associated to a generic step of the iterative algorithm.

The equivalence of named functions also imposes constraints on names and groups of partitions. Namely, the number of names and the group of the equivalence classes in $\ker H_1$ and $\ker H_2$ must be the same, as specified by the definition below.

Definition 3.4 (Equivalence of named functions) *Let* H_1 *and* H_2 *be two named functions such that* $\mathrm{dom}_{H_1} = \mathrm{dom}_{H_2}$. *We say that* H_1 and H_2 are equivalent (written $H_1 \eqsim H_2$) *if, and only if, the following conditions hold:*

- $\ker \mathrm{h}_{H_1} = \ker \mathrm{h}_{H_2}$;
- $\forall q \in \mathrm{dom}_{H_1}.|\mathrm{h}_{H_1}(q)|_{\mathrm{cod}_{H_1}} = |\mathrm{h}_{H_2}(q)|_{\mathrm{cod}_{H_2}}$;
- $\forall q \in \mathrm{dom}_{H_1}.\mathcal{G}_{\mathrm{cod}_{H_1}}(\mathrm{h}_{H_1}(q)) = \mathcal{G}_{\mathrm{cod}_{H_2}}(\mathrm{h}_{H_2}(q))$.

Let us remark that Definition 3.4 does not imply that H_1 and H_2 are the same named function when $H_1 \eqsim H_2$ because the underlying functions h_{H_1} and h_{H_2} can differ each other.

Proposition 3.1 $\eqsim$ *is an equivalence relation.*

Proof. Trivial. □

The following relation on named functions will be exploited to compare named functions.

Definition 3.5 ($\preceq$) *Let* H_1 *and* H_2 *be two named functions such that* $\mathrm{dom}_{H_1} = \mathrm{dom}_{H_2}$. *Relation* $H_1 \preceq H_2$ *holds if, and only if,*

- *partition* $Q_{\ker H_1}$ *is coarser than* $Q_{\ker H_2}$;
- $\forall A \in Q_{\ker H_1}.\forall B \in Q_{\ker H_2}.A \cap B \neq \emptyset \Rightarrow |A|_{\ker H_1} \leq |B|_{\ker H_2}$;
- $\forall A \in Q_{\ker H_1}.\forall B \in Q_{\ker H_2}.\forall q \in A \cap B.\Sigma_{H_1}(q) \subseteq \Sigma_{H_2}(q)$.

Proposition 3.2 $\preceq$ *is a pre-order.*

Proof. Trivial. $\square$

Proposition 3.3 *Let H_1 and H_2 be two named functions, then*

$$H_1 \preceq H_2 \ \wedge\ H_2 \preceq H_1 \quad \Longrightarrow \quad H_1 \eqsim H_2 \tag{1}$$

Proof. The first two conditions of Definition 3.5 and the hypothesis of (1) trivially implies the first two conditions of Definition 3.4. It remains to prove that the hypothesis implies the last condition of Definition 3.4.

Assume that there is $q \in \mathrm{dom}_{H_1}$ such that $\mathcal{G}_{\mathrm{cod}_{H_1}}(\mathrm{h}_{H_1}(q)) \neq \mathcal{G}_{\mathrm{cod}_{H_2}}(\mathrm{h}_{H_2}(q))$. Then, for all $\sigma \in \Sigma_{H_1}(q)$,

$$\mathcal{G}_{\mathrm{cod}_{H_1}}(\mathrm{h}_{H_1}(q)); \sigma \neq \mathcal{G}_{\mathrm{cod}_{H_2}}(\mathrm{h}_{H_2}(q)); \sigma. \tag{2}$$

Notice that $H_1 \preceq H_2 \ \wedge\ H_2 \preceq H_1$ and Definition 3.5 imply $\Sigma_{H_1}(q) = \Sigma_{H_2}(q)$; and conditions on named functions imply that $\mathcal{G}_{\mathrm{cod}_{H_i}}(\mathrm{h}_{H_i}(q)); \sigma = \Sigma_{\mathrm{h}_{H_i}}(q)$ (for $i = 1, 2$), that contradicts (2). $\square$

Composition of named functions is defined as expected.

Definition 3.6 (Composition of named functions) *Let H, K : `NF` be two named functions. We say that H and K are* composable *if, and only if,* $\mathrm{cod}_H = \mathrm{dom}_K$*, thence the* composition of H and K *is the named function $H; K$ such that* $\mathrm{dom}_{H;K} = \mathrm{dom}_H$*,* $\mathrm{cod}_{H;K} = \mathrm{cod}_K$*,* $\mathrm{h}_{H;K} = \mathrm{h}_H; \mathrm{h}_K$ *and* $\Sigma_{H;K} = \lambda q \in \mathrm{dom}_{H;K}.\Sigma_K(\mathrm{h}_H(q)); \Sigma_H(q)$*.*

Proposition 3.4 *Let H and K be two composable named functions. Conditions 1,2 and 3 in Definition 3.2 hold for $H; K$.*

Proof. We assume fixed a generic $q \in \mathrm{dom}_{H;K}$ and $\sigma \in \Sigma_{H;K}(q)$. First, recall that $\Sigma_{H;K}(q) = \Sigma_K(\mathrm{h}_H(q)); \Sigma_H(q)$, therefore, there are $\sigma_1 \in \Sigma_K(\mathrm{h}_H(q))$ and $\sigma_2 \in \Sigma_H(q)$, such that $\sigma = \sigma_1; \sigma_2$.

Condition 1 We must prove that $\mathcal{G}_{\mathrm{cod}_{H;K}}(\mathrm{h}_{H;K}(q)); \sigma = \Sigma_{H;K}(q)$. We first consider $\subseteq$:

$$\begin{array}{ll}
\mathcal{G}_{\mathrm{cod}_{H;K}}(\mathrm{h}_{H;K}(q)); \sigma & \\
= & \text{Definition 3.6 and } \sigma = \sigma_1; \sigma_2 \\
\mathcal{G}_{\mathrm{cod}_K}(\mathrm{h}_K(\mathrm{h}_H(q))); (\sigma_1; \sigma_2) & \\
= & \text{associativity of composition} \\
(\mathcal{G}_{\mathrm{cod}_K}(\mathrm{h}_K(\mathrm{h}_H(q))); \sigma_1); \sigma_2 & \\
= & \text{Definition 3.2 (condition 1 on } K\text{)} \\
\Sigma_K(\mathrm{h}_H(q)); \sigma_2 & \\
\subseteq & \text{def. of } \Sigma_{H;K} \\
\Sigma_{H;K}(q). &
\end{array}$$

For the reverse inclusion, we must prove that σ can be written as composition of a permutation in $\mathcal{G}_{\mathrm{cod}_{H;K}}(\mathrm{h}_{H;K}(q))$ and a $\sigma' \in \Sigma_{H;K}(q)$. By Definition 3.2, $\sigma_1 \in \mathcal{G}_{\mathrm{cod}_K}(\mathrm{h}_K(\mathrm{h}_H(q))); \sigma_1'$, for a suitable $\sigma_1' \in \Sigma_K(\mathrm{h}_H(q))$; this proves the inclusion.

Condition 2 We must prove that $\sigma; \mathcal{G}_{\mathrm{dom}_{H;K}}(q) \subseteq \Sigma_{H;K}(q)$, namely, by Definition 3.6, $\sigma; \mathcal{G}_{\mathrm{dom}_H}(q) \subseteq \Sigma_K(\mathrm{h}_H(q)); \Sigma_H(q)$:

$$\begin{array}{ll}
(\sigma_1; \sigma_2); \mathcal{G}_{\mathrm{dom}_H}(q) & \\
= & \text{associativity of composition} \\
\sigma_1; (\sigma_2; \mathcal{G}_{\mathrm{dom}_H}(q)) & \\
\subseteq & \text{Definition 3.2 (condition 2 on } H\text{)} \\
\sigma_1; (\Sigma_H(q)) &
\end{array}$$

that completes the proof.

Condition 3 This trivially holds because injectivity is preserved by composition and, by definition of named function, both $\Sigma_K(\mathrm{h}_H(q))$ and $\Sigma_H(q)$ contains only injective functions and, for any q, $\Sigma_{H;K}(q) = \Sigma_K(\mathrm{h}_H(q)); \Sigma_H(q)$, by Definition 3.6. $\Box$

The above proposition and Definition 3.6 ensure that the composition of named functions is a named function.

3.2 HD-automata for π-calculus

In order to maintain the presentation as simple as possible, this section collects the definitions specific for π-calculus (with early semantics), instead of specifying the general framework independently from a specific calculus. However, we remark that Mihda is not specific for the π-calculus but it can handle the general case.

3.2.1 Bundles for the π-calculus

We represent π-calculus labels with the enumeration type `L` defined as

$$\mathtt{L} \triangleq \mathtt{TAU}, \mathtt{IN}, \mathtt{OUT}, \mathtt{BOUT}, \mathtt{BIN}$$

(that we consider ordered by their positions). All but label `BIN` have a corresponding label in the transition system of π-calculus reported in Section 2.3. Transitions labelled by `BIN` correspond to π-calculus input transitions whose object name is a fresh name, namely, the object name does not appear in the free names of the agent performing the transition. In the following, we call *bound input* such transitions, while *bound transitions* either are bound output or bound input transitions. Non bound transitions are called free transitions. Notice that, according to the early semantics of π-calculus, there is an infinite number of bound transitions out of a state of the form $xy.P$ or $(\nu\, y)\bar{x}y.P$, however they all are equivalent up to renaming of the fresh name. Therefore, they can be represented by means of a single `BIN` or `BOUT` transition in the HD-automata for π-calculus.

Since names are local to states, it is necessary to specify how label names are related to names of states. Let $|\_|$ be the weight map associating to each π-label l a set of as many indexes as are the names l refers to. The weight map is defined as follows:

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

According to $|\_|$, no name is associated to synchronisation label `TAU`, two names (the subject and the object of the transition) are associated to `IN` and `OUT`, whereas one name is associated to `BOUT` and `BIN` labels.

A transition system (or an automaton) can be described coalgebraically. Intuitively, the coalgebra corresponding to the transition system is a function mapping each state to the set of transitions out of that state. Transitions for nominal calculi also specify how names of labels and target state are related to names of the starting state; they can be defined by the type `qd` of *quadruples* as

$$\mathtt{qd} \triangleq \prod_{D:\mathtt{NS}} \sum_{q \in D:} \sum_{l:\mathtt{L}} \sum_{\pi:|l| \to \omega} \mathtt{N}(|q|_D) \to \omega.$$

Quadruples are built on named sets and can be seen as 4-tuples. It is worth to assign a name to quadruples' projections, in order to enhance readability. Let $D : \mathtt{NS}$ be a named set and $t : \mathtt{qd}(D)$ be a quadruple on D, then

$$\tau_t \triangleq \mathrm{I}(t), \quad \ell_t \triangleq \mathrm{I}(\mathrm{II}(t)), \quad \pi_t \triangleq \mathrm{I}(\mathrm{II}(\mathrm{II}(t))), \quad \sigma_t \triangleq \mathrm{II}(\mathrm{II}(\mathrm{II}(t))).$$

A quadruple t over a named set D represents a transition to state τ_t with label ℓ_t. Each transition is equipped with two functions: π_t and σ_t that respectively map indexes of ℓ_t and names of τ_t to suitable names (of the source state of the transition). Both π_t and σ_t are necessary for establishing a relationship between indexes in labels and names local to states or between names of different states.

Remark 3.2 *In the case of* `BOUT` *and* `BIN` *labels, we must relate the subject name of the transition to a name in the source state, while (as will be clearer later) the newly generated name (*`BOUT`*) and the fresh name acquired by an input transitions (*`BIN`*) do not appear in* π_t*. The information about bound names is given in* σ_t*. In fact, if* t *is a bound transition,* σ_t *maps the bound name (of* τ_t*) to* 0*. In this respect,* 0 *should not be considered as a name, but as a placeholder that signals that a name of the target state must be considered new. No name is mapped to* 0 *by* σ_t *when* ℓ_t *is* `TAU`*,* `IN` *or* `OUT`*.*

Quadruples can be totally ordered using the lexicographic order $\sqsubseteq$, where $t \sqsubseteq t'$ if, and only if,

$$\tau_t \lesssim_D \tau_{t'} \wedge (\tau_t = \tau_{t'} \Rightarrow \ell_t \leq \ell_{t'} \wedge (\ell_t = \ell_{t'} \Rightarrow \pi_t \trianglelefteq \pi_{t'} \wedge (\pi_t = \pi_{t'} \Rightarrow \sigma_t \trianglelefteq \sigma_{t'}))).$$

The intuition is that $\sqsubseteq$ exploits the orders of the underlying components for imposing an order relation on quadruples. Such order will become useful later to define the action of the functor over HD-automata for π-agents.

We call *bundle* the collection of transitions out of a state. Bundles are described by type `B` below:

$$\mathtt{B} \stackrel{\triangle}{=} \sum_{D:\mathtt{NS}} \wp_{\text{fin}}(\mathtt{qd}(D)).$$

A bundle over a named set D is a pair whose first component is D and the second component is a set of quadruples on D. Analogously to what done for previous types, we assign names to the components of a bundle $\beta : \mathtt{B}$; the *support of* β is $\mathrm{I}(\beta)$ and is denoted by D_β, while the *step of* β, denoted by $\mathcal{S}_\beta$, is $\mathrm{II}(\beta)$.

We can lift $\sqsubseteq$ to bundles (over the same support). Indeed, $\sqsubseteq$ induces the following order on bundles.

Definition 3.7 (Bundles ordering) *Let $\beta_1, \beta_2 : \mathtt{B}$ be two bundles such that $D_{\beta_1} = D_{\beta_2}$ and let t_i be the minimal quadruple in $\mathcal{S}_{\beta_i}$ (for $i = 1, 2$). We say that β_1 is* smaller *than β_2 (and write $\beta_1 \precsim \beta_2$) if, and only if,*

- *either $\mathcal{S}_{\beta_1}$ is empty,*
- *or $t_1 \sqsubseteq t_2$ and $t_1 \neq t_2$,*
- *or else $t_1 = t_2$ and $\langle D_{\beta_1}, \mathcal{S}_{\beta_1} \setminus \{t_1\}\rangle \precsim \langle D_{\beta_1}, \mathcal{S}_{\beta_2} \setminus \{t_2\}\rangle$.*

Basically, $\precsim$ corresponds to the lexicographic order on the second components of the bundles.

3.2.2 Operations

This section reports the auxiliary operations on bundles that are used in the specification of the functor for coalgebras over the π-calculus.

We start by showing how bundles can be *casted* to named sets. Since Definition 3.7 provides an order on bundles, it suffices to define the names and the group of a bundle.

The names of a bundle are those names that "appear" in the ranges of π_t and σ_t of its quadruples t. If we define the function

$$(\!|\_|\!)_{\_} \stackrel{\triangle}{=} \lambda D : \mathtt{NS}.\lambda t : \mathtt{qd}(D). \bigcup_{\substack{n\,:\,|\ell_t| \\ m\,:\,\mathtt{N}(|\tau_t|_D)}} \{\pi_t(n), \sigma_t(m)\}$$

(we write $(\!|\, t \,|\!)_D$ instead of $(\!|\,|\!)(D)(t)$); then names of a bundle can be simply computed by the function

$$\{\!|\_|\!\} \stackrel{\triangle}{=} \lambda\beta : \mathtt{B}. \bigcup_{t \in \mathcal{S}_\beta} (\!|\, t \,|\!)_{D_\beta} \;\setminus\; \{0\},$$

while $\lfloor \beta \rfloor$ denotes the cardinality of $\{\!| \beta |\!\}$ and is the weight function on bundles.

When the minimisation algorithm will be introduced (Section 3.3), we will see that bundles also play the rôle of states along the iterations of the algorithm. Hence, the names of a bundle obey the same conventions of names of states and, according to Remark 3.2, the natural number 0 should not be considered a name of the bundle; for this reason it is removed when appearing in some transition.

Let β be a bundle and ρ be a permutation of $\{\!| \beta |\!\}$, $\beta\rho$ denotes the bundle whose support is D_β and whose step is

$$\{\langle \tau_t, \ell_t, \pi_t; \rho, \sigma_t; \rho \rangle | \text{ is a permutation of } \{\!| \beta |\!\} \ \wedge \ t \in \mathcal{S}_\beta\}.$$

The symmetry of β, $Gr(\beta)$, consists of all the bijections of $\{\!| \beta |\!\}$ that leave (the step of) β unchanged, namely

$$Gr(\beta) = \{\rho | \rho \text{ is a permutation of } \{\!| \beta |\!\} \ \wedge \ \ \mathcal{S}_\beta = \mathcal{S}_{\beta\rho}\}.$$

The most important operation on bundles is *normalisation.* It is necessary because (*i*) we must establish a canonical way of choosing the step component of a bundle among a number of different equivalent ways; (*ii*) more importantly, *redundant* input transitions must be removed. Redundancy is strictly connected to the concept of *active names.* A name n is *inactive* for an agent P whenever P is bisimilar to $(\nu\, n)P$, otherwise it is *active* for P. Intuitively, a name is inactive if it will not be used in the future transitions of the process. In general, deciding whether a name is active or not is as difficult as deciding the bisimilarity of two processes. Redundant transitions are free input transitions where the passed name is inactive in the source of the transition.

The importance of redundancy becomes evident when we try to establish the equivalence of processes that have different numbers of free names. For instance, the following π-agents:

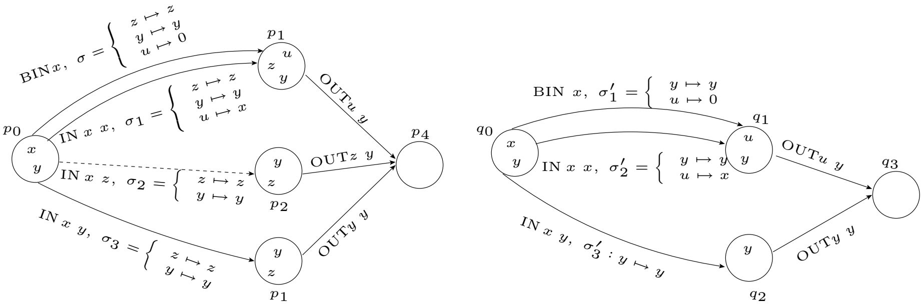
$$P \overset{\triangle}{=} x(u).\nu\, v(\bar{v}z + \bar{u}y), \qquad\qquad Q \overset{\triangle}{=} x(u).\bar{u}y,$$

are bisimilar only if, for any name substituted for u, their continuations remain bisimilar. However, P has a free input transition which corresponds to passing name z while Q has not. Thus, unless this transition is recognised as redundant and removed, the automata for P and Q would not be bisimilar. The transition is redundant since it is *dominated* by the bound input transition of P, where a fresh name is passed. Being z inactive in P, passing name z is like passing a fresh name.

Redundant transitions occur when HD-automata are built from π-agents. During this phase, it is not possible to decide which free input transitions are required, and which transitions are redundant[3]. The solution to this problem consists of adding all the free input transitions when HD-automata are built, and to exploit a reduction function (at every step of the partition algorithm) to remove those that are unnecessary.

---

[3] In general, to decide whether a free input transition is redundant or not is equivalent to decide whether a name is active or not; therefore, it is as difficult as deciding bisimilarity.


p_0
x
y
$\mathrm{BIN}\,x,\ \sigma = \left\{ \begin{array}{l} z \mapsto z \\ y \mapsto y \\ u \mapsto 0 \end{array} \right.$
$\mathrm{IN}\ x\ x,\ \sigma_1 = \left\{ \begin{array}{l} z \mapsto z \\ y \mapsto y \\ u \mapsto x \end{array} \right.$
$\mathrm{IN}\ x\ z,\ \sigma_2 = \left\{ \begin{array}{l} z \mapsto z \\ y \mapsto y \end{array} \right.$
$\mathrm{IN}\ x\ y,\ \sigma_3 = \left\{ \begin{array}{l} z \mapsto z \\ y \mapsto y \end{array} \right.$
p_1
u
z
y
p_2
y
z
p_1
y
z
$\mathrm{OUT}u\ y$
$\mathrm{OUT}z\ y$
$\mathrm{OUT}y\ y$
p_4
q_0
x
y
$\mathrm{BIN}\ x,\ \sigma'_1 = \left\{ \begin{array}{l} y \mapsto y \\ u \mapsto 0 \end{array} \right.$
$\mathrm{IN}\ x\ x,\ \sigma'_2 = \left\{ \begin{array}{l} y \mapsto y \\ u \mapsto x \end{array} \right.$
$\mathrm{IN}\ x\ y,\ \sigma'_3 : y \mapsto y$
q_1
u
y
q_2
y
$\mathrm{OUT}u\ y$
$\mathrm{OUT}y\ y$
q_3

(a) HD-automaton for P (b) HD-automaton for Q

Figure 2: Redundant transitions

Figure 2 reports the HD-automata corresponding to P and Q (in Section 3.2.4 we formally define how HD-automata can be associated to π-agents)[4]. Transition with label `IN` $x\ z$ (drawn with dashed lines) is redundant in the automata of P. States p_0 and q_0 in Figure 2 are bisimilar because transition from p_0 to p_2 is redundant since z is inactive in p_0. Transition $\langle p_2, \mathtt{IN}, xz, \sigma_2 \rangle$ expresses exactly the behaviour of the bound input, except that a free redundant input transition is used rather than a bound one. In other words, when the bisimulation game is played, the free input transition to p_2 from p_0 plays the same rôle as the bound input that "assigns" a name to u that is "not known" in p_1. This means that the automaton for P is equivalent to that obtained by removing the redundant input transition which is exactly the automaton for Q.

As will be clearer in Section 4, during the iterations of the minimisation algorithm the sets of redundant transitions of bundles decrease. When the iterative construction terminates, only those free inputs that are really redundant will be removed from the bundles.

Notation 3.1 *Given a function f, in the rest of the paper, we will use the following notations:*

- $f[x \mapsto y]$ *abbreviates* $\lambda u.if\ u = x\ then\ y\ else\ f(u)$.
- $f|_B$ *is the restriction of* f *to the set* $B \subseteq \mathrm{dom}_f$.

The normalisation of a bundle is done in different steps. First, the bundle is reduced by removing all the possibly redundant input transitions using the function *red*:

$$red \overset{\triangle}{=} \lambda\beta : \mathtt{B}.\langle D_\beta,\ \ \mathcal{S}_\beta \setminus \{t \in \mathcal{S}_\beta | dominated(\beta)(t)\}\rangle$$

---

[4] The convention adopted for names in the π-component of transitions in Figure 2 is that the name in the i-th position is $\pi(i)$. For instance, sequence $x\ y$ stands for $\pi(1) = x$ and $\pi(2) = y$.

where, *dominated* expresses the condition for quadruples' redundancy (for the early semantics of π-calculus) and it is defined as follows:

$$
\begin{aligned}
\mathit{dominated} \triangleq{} & \lambda\beta : \mathtt{B}.\lambda t : \mathtt{qd}(D_\beta). \\
& \quad \ell_t = \mathtt{IN} \ \wedge\ \langle \tau_t, \mathtt{BIN}, \pi_t|_{\{1\}}, \sigma_t[\sigma_t^{-1}(\pi_t(2)) \mapsto 0]\rangle \in \mathcal{S}_\beta.
\end{aligned}
$$

The intuition is that a transition t is dominated by a bundle β when it is a free input transition and β contains a bound input transition to the same target of t (on the same channel) and such that the object name of t is mapped on to the 0 name in the bound input transition.

It is important to observe that dominated transition are *not* redundant transitions. Dominated transitions are only used to compute the *active names of a bundle*, that are those names of the reduced bundle defined by:

$$
\mathit{an} \quad \triangleq \quad \lambda\beta : \mathtt{B}.\{\!|\, \mathit{red}(\beta) \,|\!\}.
$$

Let us remark that this is a different concept than active names of a process. Active names of a bundle do not involve any notion of "future" behaviour; they are characterised in terms of a local property of the bundle.

We use function *rem_in* to remove input transitions from a bundle if their object names are not in a given set of names. Function *rem_in* is defined as follows:

$$
\mathit{rem\_in} \triangleq \lambda\beta : \mathtt{B}.\lambda N : \wp_{\mathrm{fin}}(\omega).\mathcal{S}_\beta \setminus \{t \in \mathcal{S}_\beta | \ell_t = \mathtt{IN} \ \wedge\ \pi_t(2) \notin N\}.
$$

Finally, the function to normalise bundles is defined as follows:

$$
\begin{aligned}
\mathit{norm} \quad \triangleq{} & \lambda\beta : \mathtt{B}. \\
& \quad \mathit{let}\ \beta' = \mathit{rem\_in}\ \beta\ (\mathit{an}(\beta)) \\
& \quad \mathit{in}\ \mathit{min}_{\sqsubseteq}(\beta'\theta \mid \theta : \mathit{an}(\beta) \to \mathtt{N}(|\mathit{an}(\beta)|) \text{ is a bijective substitution}).
\end{aligned}
$$

We also define θ_β as the bijective substitution for which the minimal is achieved, i.e. $(\mathit{rem\_in}\ \beta\ (\mathit{an}(\beta)))\theta_\beta = \mathit{norm}\ \beta$.

Basically, *norm*, applied to a bundle β, filters those input transitions that are dominated in β and whose object names are no longer active names of the (reduced) bundle. The remaining transition are collected in a bundle and the substitution which makes it minimal is applied to it.

We have the following results.

Lemma 3.1 *If $\beta, \beta' : \mathtt{B}$ are such that $\mathcal{S}_\beta \subseteq \mathcal{S}_{\beta'}$, and let $\beta_1 = \mathit{norm}\ \beta$ and $\beta'_1 = \mathit{norm}\ \beta'$ then $\mathcal{S}_{\beta_1}; \theta^{-1}(\beta) \subseteq \mathcal{S}_{\beta'_1}; \theta^{-1}(\beta')$.*

Proof. Trivial. □

Lemma 3.2 *For each $\beta : \mathtt{B}$, $\mathcal{S}_{\mathit{norm}\ \beta} \subseteq \mathcal{S}_\beta; \theta^{-1}(\beta)$.*

Proof. By construction, *norm* β is the bundle obtained by removing dominated quadruples from β and by renaming its names through $\theta^{-1}(\beta)$. □

Intuitively, Lemma 3.2 states that β "includes" the bundle resulting from its normalisation. Sometimes we write $\beta_1 \subseteq \beta_2$ instead of $\mathcal{S}_{\beta_1} \subseteq \mathcal{S}_{\beta_2}$.

3.2.3 The functor for π-calculus

This section, by taking advantage of the constructions previously introduced, details the functor T for the π-calculus. As usual [21], we represent T as a pair (T_1, T_2), where T_1 maps objects to objects, while T_2 maps morphisms to morphisms. Map T_1 is defined as in Figure 3. When applied to a named set A, it returns a named set whose components are described below:

- the underlying set is characterised by the type of bundles over A;
- the order relation is the order on bundles induced by the order of A (Definition 3.7);
- the weight function is the function that gives the number of names appearing in the quadruples of the bundle;

```
$T_1 \overset{\triangle}{=} \lambda A : \mathtt{NS}.$
    $\langle\ B = Q_A \times \wp_{\mathrm{fin}}(\mathtt{qd}(Q_A)),$
      $\lesssim\ = \lambda(\beta, \beta') : B \times B.$
        $if\ \mathcal{S}_\beta = \emptyset$
          $then\ \mathtt{tt}$
          $else$
            $let\ t = min_{\sqsubseteq}(\mathcal{S}_\beta)\ in$
            $let\ t' = min_{\sqsubseteq}(\mathcal{S}_{\beta'})\ in$
              $t \sqsubseteq t' \wedge (t = t' \implies \mathcal{S}_\beta \setminus \{t\} \lesssim \mathcal{S}_{\beta'} \setminus \{\beta'\}),$
      $|\_| = \lambda\beta : B.\ \lfloor\beta\rfloor,$
      $\lambda\beta : B.Gr(\beta)$
    $\rangle$
```

Figure 3: Functor on named sets

Figure 4 illustrates the actions of the bundle on named functions through the map T_2. The named function resulting from applying T_2 to H is a function such that

```
$T_2 \overset{\triangle}{=} \lambda H : \mathtt{NF}.$
    $let\ S = T_1(\mathrm{dom}_H)$
    $and\ D = T_1(\mathrm{cod}_H)$
    $and\ h' = \lambda\beta \in S.$
        $\langle \mathrm{cod}_H,\ \{\langle \mathrm{h}_H(q), l, \pi, \sigma'; \sigma\rangle\} \mid \langle q, l, \pi, \sigma\rangle \in \mathcal{S}_\beta \ \wedge\ \sigma' \in \Sigma_{H(q)}\}\rangle$
    $and\ h = \lambda\beta \in S.norm\ h'(\beta)$
    $and\ \Sigma = \lambda\beta \in S.\{\rho; \theta^{-1}(h'(\beta)) | \rho \in Gr(h(\beta))\}$
    $in\ \langle S, D, h, \Sigma\rangle$
```

Figure 4: Functor on named functions

- the domain is obtained by applying T_1 to dom_H;
- the codomain is obtained by applying T_1 to cod_H;

- the function $\mathrm{h}_{T_2(H)}$ maps each bundle in the domain to a normalised bundle in the codomain;
- for each bundle β, the set of its name correspondences is obtained by composing symmetry $Gr(\mathrm{h}_{T_2(H)}(\beta))$ with $\theta^{-1}(\beta)$ to undo the normalisation of names performed by computing *norm*.

Map T_2 is *functorial*, i.e. it preserves composition and identity:

Proposition 3.5 *Let $H : \mathtt{NF}$, $K : \mathtt{NF}$ be two composable named functions.*

$$T_2(H;K) = T_2(H);T_2(K). \tag{3}$$

PROOF. First, we prove that both sides of equation 3 have the same domain and codomain.

$$\begin{array}{lcll}
\mathrm{dom}_{T_2(H;K)} & = & T_1(\mathrm{dom}_{H;K}) & \text{by def. of } T_2 \\
 & = & T_1(\mathrm{dom}_H) & \text{by def. of ';'} \\
\mathrm{dom}_{T_2(H);T_2(K)} & = & \mathrm{dom}_{T_2(H)} & \text{by def. of ';'} \\
 & = & T_1(\mathrm{dom}_H) & \text{by def. of } T_2.
\end{array}$$

A similar proof shows that $\mathrm{cod}_{T_2(H;K)} = \mathrm{cod}_{T_2(H);T_2(K)}$.

Second, we show that $\mathrm{h}_{T_2(H;K)} = \mathrm{h}_{T_2(H);T_2(K)}$. By definition 3.6 and the definition of T_2, we have, for any $\beta \in \mathrm{dom}_{T_2(H)}$,

$$\mathrm{h}_{T_2(H);T_2(K)}(\beta) \quad = \quad \mathrm{h}_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta)) \quad = \quad \mathit{norm}\ \langle \mathrm{cod}_{T_2(K)}, Q_\beta \rangle$$

where Q_β is the set of quadruples $\langle \mathrm{h}_K(\beta'), \ell', \pi', \bar{\sigma}; \sigma' \rangle$ such that $\langle \beta', \ell', \pi', \sigma' \rangle \in \mathcal{S}_{\mathrm{h}_{T_2(H)}(\beta)}$ and $\bar{\sigma} \in \Sigma_K(\beta')$, while the step of $\mathrm{h}_{T_2(H)}(\beta)$ is the step of the bundle

$$\mathit{norm}\ \langle \mathrm{cod}_{T_2(H)}, \bigcup_{\langle q,\ell,\pi,\hat{\sigma} \rangle \in \mathcal{S}_\beta} \{ \langle \mathrm{h}_H(q), \ell, \pi, \bar{\sigma}; \hat{\sigma} \rangle | \bar{\sigma} \in \Sigma_H(q) \} \rangle,$$

and, by Lemma 3.2,

$$\mathcal{S}_{\mathrm{h}_{T_2(H)}(\beta)} \subseteq \bigcup_{\langle q,\ell,\pi,\hat{\sigma} \rangle \in \mathcal{S}_\beta} \{ \langle \mathrm{h}_H(q), \ell, \pi, \bar{\sigma}; \hat{\sigma} \rangle | \bar{\sigma} \in \Sigma_H(q) \}. \tag{4}$$

Let us now consider $\mathrm{h}_{T_2(H;K)}$:

$$\mathrm{h}_{T_2(H;K)}(\beta) = \mathit{norm}\ \langle \mathrm{cod}_{T_2(H;K)}, \bigcup_{\langle q,\ell,\pi,\hat{\sigma} \rangle \in \mathcal{S}_\beta} \{ \langle \mathrm{h}_{H;K}(q), \ell, \pi, \bar{\sigma}; \hat{\sigma} \rangle | \bar{\sigma} \in \Sigma_{H;K}(q) \} \rangle,$$

hence, $\mathrm{h}_{T_2(H;K)}(\beta) = \mathit{norm}\ \langle \mathrm{cod}_{T_2(K)}, \mathcal{S} \rangle$ where, $\mathcal{S}$ contains all quadruples of the form $\langle \mathrm{h}_K(\mathrm{h}_H(q)), \ell, \pi, \bar{\sigma}; \hat{\sigma} \rangle$ such that $\langle q, \ell, \pi, \hat{\sigma} \rangle \in \mathcal{S}_\beta$ and $\bar{\sigma} \in \Sigma_K(\mathrm{h}_H(q)); \Sigma_H(q)$. This, together with (4), implies (for any β) $Q_\beta \subseteq \mathcal{S}_{\mathrm{h}_{T_2(H;K)}(\beta)}$ and we have that $\mathrm{h}_{T_2(H);T_2(K)} \subseteq \mathrm{h}_{T_2(H;K)}(\beta)$ follows by Lemma 3.1.

In order to prove $\mathrm{h}_{T_2(H);T_2(K)} = \mathrm{h}_{T_2(H;K)}(\beta)$, it remains to show that $\mathrm{h}_{T_2(H;K)}(\beta) \subseteq \mathrm{h}_{T_2(H);T_2(K)}$. If $t \in \mathcal{S}_{\mathrm{h}_{T_2(H;K)}}$ and the label of t is not `IN`, then $t \in \mathcal{S}_{\mathrm{h}_{T_2(H);T_2(K)}}$. Since t can be in $\mathcal{S}_{\mathrm{h}_{T_2(H;K)}} \setminus \mathcal{S}_{\mathrm{h}_{T_2(H);T_2(K)}}$ only if t is dominated in $\mathcal{S}_{\mathrm{h}_{T_2(H);T_2(K)}}$, in which case there is a `BIN` transition in $\mathcal{S}_{\mathrm{h}_{T_2(H);T_2(K)}(\beta)}$ that dominates t and, recalling that $\mathcal{S}_{\mathrm{h}_{T_2(H);T_2(K)}(\beta)} \subseteq \mathcal{S}_{\mathrm{h}_{T_2(H;K)}(\beta)}$, then t would be dominated also in $\mathcal{S}_{\mathrm{h}_{T_2(H;K)}(\beta)}$. This contradicts the hypothesis that $t \in \mathcal{S}_{\mathrm{h}_{T_2(H;K)}(\beta)}$.

Finally, we show that $\Sigma_{T_2(H;K)} = \Sigma_{T_2(H);T_2(K)}$. By definition of composition of named functions we have

$$\Sigma_{T_2(H);T_2(K)}(\beta) = \Sigma_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta)); \Sigma_{T_2(H)}(\beta),$$

hence

$$\Sigma_{T_2(H);T_2(K)}(\beta) = (Gr(\mathrm{h}_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta))))\theta^{-1}(\mathrm{h}_{T_2(H)}(\beta))); (Gr(\mathrm{h}_{T_2(H)}(\beta))\theta^{-1}(\beta)).$$

Moreover, $\theta^{-1}(\mathrm{h}_{T_2(H)}(\beta))$ is the identity (because $\mathrm{h}_{T_2(H)}(\beta)$ is a normalised bundle, by definition of T_2) then we have

$$\begin{aligned}\Sigma_{T_2(H);T_2(K)}(\beta) &= Gr(\mathrm{h}_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta))); (Gr(\mathrm{h}_{T_2(H)}(\beta)); \theta^{-1}(\beta)) \\ &= \bigcup_{\sigma \in \Sigma_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta))} Gr(\mathrm{h}_{T_2(K)}(\mathrm{h}_{T_2(H)}(\beta))); \sigma \qquad (5)\end{aligned}$$

(equality (5) by associativity of composition). By Definition 3.2, we conclude from (5) that $\Sigma_{T_2(H);T_2(K)}(\beta) = \Sigma_{T_2 H;K}(\beta)$. $\square$
Proposition 3.5 shows that T_2 preserves composition. It remain to prove that identities are also preserved. Given a named set $A : \texttt{NS}$, $\mathrm{id}_A : \texttt{NF}$ is the named functions such that $\mathrm{dom}_{idA} = A$ and $\mathrm{h}_{\mathrm{id}_A}$ is the identity on Q_A and, $\Sigma_{\mathrm{id}_A}(q)$ only contains the identity on names of q, for any $q : Q_A$. Thence, it is trivial to see that, by construction, $T_2(\mathrm{id}_A)$ is the identity named function on $T_1(A)$, that, with Proposition 3.5 gives the functoriality of T_2.

3.2.4 From π-calculus to HD-automata

Following [15], we now construct the HD-automaton corresponding to the early semantics of the π-calculus. We assume to have a function $\mathbf{n}$ that, given a π-agents P, returns a pair $\langle \bar{P}, \theta_P \rangle = \mathbf{n}(P)$, where $\bar{P}$ is the representative of the class of agents differing from P for a bijective substitution, θ_P, such that $\bar{P} = P\theta_P$ and $\overline{P\rho} = \bar{P}$, for any bijective substitution ρ. Hereafter, we consider $\mathcal{N} = \{x_0, x_1, \ldots, \}$ totally ordered by relation $\leq$, where $x_i \leq x_j$ if, and only if, $i \leq j$.

In order to have a fixed way for choosing the canonical transition (up to permutations of names) we borrow from [18] the definition of *representative transitions.*

Definition 3.8 (Representative transition) *A π-calculus transition $P \xrightarrow{\mu} P'$ is a* representative transition *if one of the following conditions applies:*

- *either $\mu = \tau$ or $\mu = \bar{x}y$;*
- *$\mu = \bar{x}(y)$ and $y = \min(\mathcal{N} \setminus \mathrm{fn}(P))$;*
- *$\mu = xy$ and $y \in \mathrm{fn}(P) \cup \{\min(\mathcal{N} \setminus \mathrm{fn}(P))\}$.*

Intuitively, a representative transition is exploited to single out a transition in a canonical way from a bunch of bound outputs (that differ only for the extruded name), and from a bunch of input transitions (that differ in the fresh name that is received from the environment).

We use only representative transitions in HD-automata that correspond to π-agents; the following lemma ensures that these transitions are enough for characterising agents' behaviour.

Lemma 3.3 (Lemma 7.6 of [18]) *If* $P \xrightarrow{xy} P'$ *(resp.* $P \xrightarrow{\bar{x}(y)} P'$*) is not a representative transition, then there is a representative transition* $P \xrightarrow{xz} P''$ *(resp.* $P \xrightarrow{\bar{x}(z)} P''$*) such that* $P'' = P'[^{z,y}/_{y,z}]$.

Let P be a π-calculus agent and let $\langle \bar{P}, \theta_P \rangle = \mathbf{n}(P)$. The coalgebraic specification of the corresponding HD-automaton of P can be easily obtained as a named function $K[P]$ with $\mathrm{dom}_{K[P]} = D[P]$ and $\mathrm{cod}_{K[P]} = T_1(D[P])$.

We first determine $Q_{D[P]}$, the set of the states, as follows:

$$Q_{D[P]} \triangleq \bigcup_{P' \in S[P]} Q_{D[P']} \cup S[P],$$

where $S[P] \triangleq \{\bar{P}\} \cup \{\overline{P'} \mid P \xrightarrow{\mu} P' \text{ is a representative transition} \wedge \mathbf{n}(P') = \langle \overline{P'}, \theta_{P'} \rangle\}$. Intuitively, $Q_{D[P]}$ is the set of (the canonical representative) processes that can be reached from P (through representative transitions). It is trivial to equip $Q_{D[P]}$ with a named set structure, indeed

- the order $\lesssim_{D[P]}$ on $Q_{D[P]}$ is the lexicographic order on processes;
- for any $q \in Q_{D[P]}$, the weight function $|q|_{D[P]}$ yields the cardinality of $\mathrm{fn}(q)$;
- for any $q \in Q_{D[P]}$, the group component $\mathcal{G}_{D[P]}(q)$ is simply the identity on $\mathrm{fn}(q)$.

Function $\mathrm{h}_{K[P]}$ associates to each state the bundle of its outgoing transitions and is defined as $\mathrm{h}_{K[P]} = \lambda q.\mathit{norm}\ \beta_q$ where

$$\beta_q = \langle Q_{D[P]}, \{t_\mu \mid q \xrightarrow{\mu} q' \text{ is a representative transition }\}\rangle$$

and, if $\langle \overline{q'}, \theta_{q'} \rangle = \mathbf{n}(q')$, quadruple t_μ is defined as follows:

$$t_\mu = \begin{cases} \langle \overline{q'}, \mathtt{TAU}, \emptyset, \theta_{q'}^{-1} \rangle, & \mu = \tau \\ \langle \overline{q'}, \mathtt{OUT}, \pi, \theta_{q'}^{-1} \rangle, & \mu = \bar{x}y,\ \pi(1) = x,\ \pi(2) = y \\ \langle \overline{q'}, \mathtt{IN}, \pi, \theta_{q'}^{-1} \rangle, & \mu = xy,\ y \in \mathrm{fn}(q),\ \pi(1) = x,\ \pi(2) = y \\ \langle \overline{q'}, \mathtt{BOUT}, \pi, \theta_{q'}^{-1}[\theta_{q'}(y) \mapsto 0] \rangle, & \mu = \bar{x}(y),\ \pi(1) = x \\ \langle \overline{q'}, \mathtt{BIN}, \pi, \theta_{q'}^{-1}[\theta_{q'}(y) \mapsto 0] \rangle, & \mu = xy,\ y \notin \mathrm{fn}(q),\ \pi(1) = x. \end{cases}$$

Finally, we define $\Sigma_{K[P]}(q) = \{\rho; \theta^{-1}(\beta_q) | \rho \in \mathcal{G}_{\mathrm{cod}_{K[P]}}(\mathrm{h}_{K[P]}(q))\}$, for any state q.

The HD-automaton obtained by this definition is a T-coalgebra by construction and it is a valid HD-automaton.

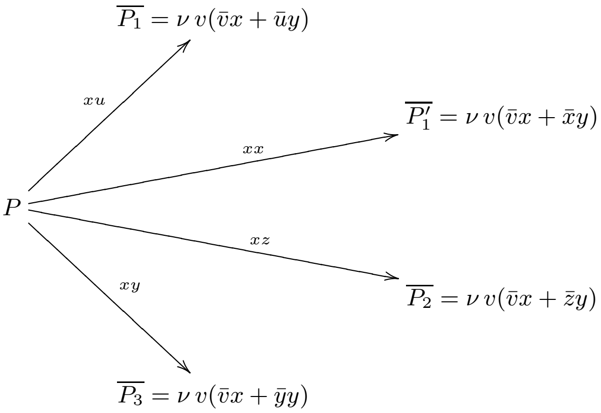
The construction above may yield infinite HD-automata. However, there are interesting classes of π-agents that generate finite HD-automata: This is the case of *finitary* π-agents. The *degree of parallelism* $\deg(P)$ of a π-calculus agent P is defined as follows:

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

Agent P is *finitary* if $\max\{\deg(P') \mid P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_i} P'\} < \infty$. In [15, 18] the following result has been proved:

Theorem 3.1 (Theorem 47 of [15]) *Let P be a finitary π-agents. Then the HD-automaton $K[P]$ is finite.*

Example 3.1 *We show how the above mapping acts by constructing the HD-automaton of agent $P \stackrel{\triangle}{=} x(u).(\nu\, v)(\bar{v}z + \bar{u}y)$ of Section 3.2.2. Let us assume that P already is the representative element with respect to $\mathbf{n}$, i.e. $\mathbf{n}(P) = \langle P, id\rangle$. Then, by definition, P has the following three representative transitions:*


$\overline{P_1} = \nu\, v(\bar{v}x + \bar{u}y)$
xu
$\overline{P_1'} = \nu\, v(\bar{v}x + \bar{x}y)$
xx
P
xz
xy
$\overline{P_2} = \nu\, v(\bar{v}x + \bar{z}y)$
$\overline{P_3} = \nu\, v(\bar{v}x + \bar{y}y)$

Finally, the remaining representative transitions are $P_1 \xrightarrow{\bar{x}y} \mathbf{0}$, $P_2 \xrightarrow{\bar{a}y} \mathbf{0}$, $P_3 \xrightarrow{\bar{z}y} \mathbf{0}$ and $P_4 \xrightarrow{\bar{y}y} \mathbf{0}$.

Notice that $\overline{P_1}$ and $\overline{P_1'}$ are the same up-to a bijective substitution hence they would yield the same state in the automaton; transition xx and xu are distinguished by a different σ in the automaton. Actually it is easy to be convinced that Figure 2(a) represents the automaton for P. Let us remark that in the automaton, the information about the freshness of name u is given by the σ-function on the bound input transition (i.e. $\sigma_1(u) = 0$), while here the new name u is the minimal name not occurring in P and is the name used in the standard representative $\overline{P_1}$ of P_1.

The minimisation algorithm introduced in Section 3.3 builds the minimal realisation $\bar{H}$ of (finite) HD-automata by constructing (an approximation of) the final coalgebra morphism. The kernel of $\bar{H}$ yields the equivalence classes where equivalent states are grouped. The active names of each state q are those in the ranges of $\Sigma_{\bar{H}}(q)$. Let us observe that Condition 1 in Definition 3.2 guarantees that all functions in $\Sigma_{\bar{H}}(q)$ have the same range.

A tight relationship can be established between π-calculus bisimilarity and the outcome of the minimisation algorithm. On the one hand, if P and Q are two (early) bisimilar π-agents then the same minimal realisation for $K[\bar{P}]$ and $K[\bar{Q}]$ is achieved by the minimisation algorithm (if $K[\bar{P}]$ and $K[\bar{Q}]$ are finite). Namely, the morphisms $\bar{H}_P$ and $\bar{H}_Q$ are equivalent in the sense of Definition 3.4, i.e. $\ker \bar{H}_P = \ker \bar{H}_Q$ which implies that corresponding classes of $\ker \bar{H}_P$ and $\ker \bar{H}_Q$ have the same number of active names and the same symmetries. It could be the case that the name correspondences in $\Sigma_{\bar{H}_P}(\bar{P})$ and $\Sigma_{\bar{H}_Q}(\bar{Q})$ are different since $\bar{P}$ and $\bar{Q}$ can have different numbers of inactive names, however $\Sigma_{\bar{H}_P}(\bar{P}); \theta_P = \Sigma_{\bar{H}_Q}(\bar{Q}); \theta_Q$ (where $\mathbf{n}(P) = \langle \bar{P}, \theta_P\rangle$ and $\mathbf{n}(Q) = \langle \bar{Q}, \theta_Q\rangle$). Indeed, P and Q must have the same set of active names otherwise they could not be bisimilar. On the other hand, if $K[P]$ and $K[Q]$ are finite and they both have $\bar{H}$ as their minimal realisation, then P and Q are bisimilar provided that $\Sigma_{\bar{H}_P}(\bar{P});\theta_P = \Sigma_{\bar{H}_Q}(\bar{Q});\theta_Q$.

3.3 The minimisation algorithm in $\lambda^{\to,\Pi,\Sigma}$

This section specifies the minimisation algorithm for HD-automata and proves that the partition refinement algorithm converges on finite HD-automata.

As pointed out in Section 3.2.4, a transition system is a HD-automaton and can be coalgebraically specified as a named function K such that $\mathrm{cod}_K = T_1(\mathrm{dom}_K)$. We now point out the minimisation algorithm. Given a T-coalgebra $K : \mathtt{NF}$ with named set A as source, we let $\perp : \mathtt{NS} = \langle unit, \lambda x, y : unit.\mathtt{tt}, \lambda x : unit.0, \emptyset\rangle$ be the *empty* named set (i.e. the named set on the vacuum type *unit* having () as its unique element). The minimisation algorithm is defined by the equations below.

$$\text{Initial approximation:} \qquad H_0 \triangleq \langle A,\ \perp,\ \lambda q : A.(),\ \lambda q : A.\emptyset\rangle \tag{6}$$

$$\text{Iterative construction:} \qquad H_{i+1} \triangleq K; T_2(H_i). \tag{7}$$

In words, all the states of automaton K are initially considered equivalent, indeed, $\ker H_0$ gives rise to a single equivalence class containing the whole dom_K. At the generic $(i+1)$-th iteration, the image through T_2 of the i-th iteration is composed with K as prescribed in (7).

The proof of the convergence of the algorithm is based on the fact that T_2 is a monotone functor over a set whose chains are finite. Hence we first show that T is monotone.

Proposition 3.6 (Monotony of T) *T_2 is monotone.*

PROOF. Let $H_i : \mathtt{NF}$ $(i = 1, 2)$ be such that $H_1 \preceq H_2$; we prove that $T_2(H_1) \preceq T_2(H_2)$. Let (for $i = 1, 2$) $S_i = \mathrm{dom}_{T_2(H_i)}$, $D_i = \mathrm{cod}_{T_2(H_i)}$, $h_i = \mathrm{h}_{T_2(H_i)}$ and $\Sigma_i = \Sigma_{T_2(H_i)}$:

- By hypothesis, $\mathrm{dom}_{H_1} = \mathrm{dom}_{H_2}$, hence, by definition of T_1, $D_1 = D_2$ and, by construction, $\lesssim_{D_1} = \lesssim_{D_2}$.
- Given $\beta, \beta' \in S_1$ such that $h_2(\beta) = h_2(\beta')$, assume, by absurd, that $h_1(\beta) \neq h_1(\beta')$. Then, by definition of T_2, there is a quadruple t in $\mathcal{S}_\beta$ (resp. in $\mathcal{S}_{\beta'}$) s.t. for any t' in $\mathcal{S}_{\beta'}$ (resp. in $\mathcal{S}_\beta$) $\langle h_1(\tau(t)), \ell(t), \sigma(t); \mu\rangle \neq \langle h_1(\tau(t')), \ell(t'), \sigma(t'); \mu\rangle$. This yields a contradiction, since there is a $t' \in \mathcal{S}_{\beta'}$ such that $\langle h_2(\tau(t)), \ell(t), \sigma(t); \mu\rangle = \langle h_2(\tau(t')), \ell(t'), \sigma(t'); \mu\rangle$ and $H_1 \preceq H_2$ implies that $h_1(\tau(t)) = h_1(\tau(t'))$.
- For all $\beta \in S_1$, $|h_i(\beta)|_{D_i} = \lfloor h_i(\beta)\rfloor$ $(i = 1, 2)$. Hence, $|h_1(\beta)|_{D_1} \leq |h_2(\beta)|_{D_2}$ by construction; indeed, for any quadruple $t \in \mathcal{S}_\beta$, $|\tau(t)|_{\mathrm{cod}_{H_1}} \leq |\tau(t)|_{\mathrm{cod}_{H_2}}$ (because $H_1 \preceq H_2$), hence the domain of the functions in $\Sigma_1(\tau(t))$ is included in the domain of the functions in $\Sigma_2(\tau(t))$, therefore, also their ranges are in the same inclusion relations.
- Let $\beta \in \mathrm{dom}_{H_1}$ be such that $\lfloor h_1(\beta)\rfloor = \lfloor h_2(\beta)\rfloor$ and by construction $(i = 1, 2)$

  $$\Sigma_i = map\ (\lambda\rho : \mathtt{N}(\lfloor h_i(\beta)\rfloor) \to \mathtt{N}(\lfloor h_i(\beta)\rfloor).\rho; \theta^{-1}(\beta))\ Gr(h_i(\beta)).$$

  By hypothesis, $\Sigma_{H_1} \subseteq \Sigma_{H_2}$, hence, by definition of T_2, by applying

  $$map\ (\lambda\mu : \Sigma_H(\beta).\langle \mathrm{h}_H(\tau_t), \ell_t, \pi_t, \sigma_t; \mu\rangle)\ \Sigma_H(\tau_t) \tag{8}$$

  to H_1 we obtaining a number of quadruples less or equal that the application of (8) to H_2. Therefore, $Gr(h_1(\beta)) \subseteq Gr(h_2(\beta))$ which imply that $\Sigma_1 \subseteq \Sigma_2$.

$\Box$

Monotony is preserved by composition of named functions:

Proposition 3.7 (Composition and $\preceq$) *Let H_1 and H_2 be two named functions such that $H_1 \preceq H_2$. For any $K : \mathtt{NF}$ if $\mathrm{cod}_K = \mathrm{dom}_{H_1} = \mathrm{dom}_{H_2}$ then $K; H_1 \preceq K; H_2$.*

PROOF.

- By definition of composition $\mathrm{dom}_{K;H_1} = \mathrm{dom}_K = \mathrm{dom}_{K;H_2}$ and the characteristic functions of $K; H_1$ and $K; H_2$ are obtained by composing the characteristic function of K with those of H_1 and H_2, respectively.
- For all $q, q' \in \mathrm{dom}_{K;H_1}$, by definition $(K; H_2)(q) = (K; H_2)(q')$ if, and only if, $H_2(K(q)) = H_2(K(q')$. Hence, $H_1(K(q)) = H_1(K(q')$ holds because $H_1 \preceq H_2$ and this implies $(K; H_1)(q) = (K; H_1)(q')$.
- We have that

$$\begin{aligned}
|q|_{\mathrm{cod}_{K;H_1}} &= |\mathrm{h}_{H_1}(\mathrm{h}_K(q))|_{\mathrm{cod}_{H_1}} && \text{by definition of composition}\\
&\leq |\mathrm{h}_{H_1}(\mathrm{h}_K(q))|_{\mathrm{cod}_{H_2}} && \text{since } H_1 \preceq H_2\\
&= |q|_{\mathrm{cod}_{K;H_2}} && \text{by definition of composition.}
\end{aligned}$$

- if $q \in \mathrm{dom}_{K;H_1}$, we have $\Sigma_{K;H_1}(q) = \Sigma_{H_1}(\mathrm{h}_K(q)); \Sigma_K(q)$ (by definition of composition); since $H_1 \preceq H_2$ we have $\Sigma_{H_1}(\mathrm{h}_K(q)) \subseteq \Sigma_{H_2}(\mathrm{h}_K(q))$ hence

$$\Sigma_{H_1}(\mathrm{h}_K(q)); \Sigma_K(q) \subseteq \Sigma_{H_2}(\mathrm{h}_K(q)); \Sigma_K(q),$$

which is equivalent to $\Sigma_{K;H_1}(q) \subseteq \Sigma_{K;H_2}(q)$.

$\Box$

Finally, we can prove the convergence of the iterative algorithm:

Theorem 3.2 (Convergence) *The iterative algorithm described by (6) and (7) is convergent on finite state automata.*

PROOF. First, observe that, by monotony of T and Proposition 3.7, maps $F_K(H) = K; T_2(H)$ is monotone. Second, for any H, $F_K(H)$ is finite. Finally, all chains in `NF` having finite domain are finite, hence, the iterative algorithm defined in (6) and (7) converges to the maximal fix-point of F_K. $\Box$

By definition, for any named function H, $T_2(H)$ differs H because their codomains always differs. However, in Theorem 3.2 we implicitly refer to the equivalence on `NF`, i.e. $\overline{\sim}$ (Definition 3.4). Relation $\overline{\sim}$ is based on the notion of kernel of named functions, hence, the fix point is a named functions such that $\hat{H} \overline{\sim} F_K(\hat{H})$. In other words, $\hat{H}$ is an automaton isomorphic to $F_K(\hat{H})$; indeed, despite the representation of the states (i.e. the type of codomains) it is possible to establish a bijective correspondence between $\mathrm{dom}_{\hat{H}}$ and $\mathrm{dom}_{F_K(\hat{H})}$ such that order, weight and group components are preserved.

4 Mihda

The algorithm in Section 3 has been specified by exploiting the parametric polymorphism of $\lambda^{\rightarrow,\Pi,\Sigma}$ in a coalgebraic framework. It remains to show that these elegant theories can be used as a basis for the design and development of effective and usable verification toolkits.

This section describes our experience in designing and implementing Mihda, a minimisation toolkit for verifying finite state mobile systems represented in the π-calculus or in other name passing calculi. The Mihda toolkit[5] cleanly separates facilities that are language-specific (parsing, transition system calculation) from those that are independent from the calculus notation (bisimulation) in order to facilitate modifications. The type system of ocaml offers all the necessary features for implementing the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification of the minimisation algorithm. The main features of ocaml exploited in our implementation are polymorphism and encapsulation.

Encapsulation is achieved by the *module system* of ocaml. The module system of ocaml is similar to the module system of ML [12, 11, 24]; its main ingredients are *signatures*, *structures* and *functors*. The module system separates the signature, a sort of interface (i.e. definition of *abstract data type*) from structures, that are the realizations and roughly are sets of types and values. A structure satisfying a given signature is said *to match* that signature and may be parameterised using *functors*, that are functions from structures to structures: An ocaml functor constructs new modules by mapping modules of a given signature on structures of other signatures.

The following example (borrowed from [13]) defines a structure and its matching signature:

```
structure S =              signature SIG =
  struct                     sig
     type t = nat              type t
     val x : t = 7             val x : t
  end;                       end;
```

If `S.t` (resp. `S.x`) is the type (resp. the value) of `S`, a functor can be defined as

```
functor F (X : SIG) : SIG =      struct S' =
  struct                           struct
    type t = X.t * X.t               type t = nat * nat
    val x : t = (X.x,X.x)            val x : nat * nat = (7,7)
  end                              end,
```

where the rightmost structure is obtained by applying `F` to `S`.

There exists a strong relationships between $\lambda^{\rightarrow,\Pi,\Sigma}$ and the module system of ocaml. On the one hand, structure `S` can be written in $\lambda^{\rightarrow,\Pi,\Sigma}$ as

$$S = \langle t : U_1 = \omega, 7 : t\rangle : \sum_{t:U_1} t,$$

which amounts to saying that sum types signatures correspond and expression with that type are the structures matching the signature. For instance, the ocaml program `S.x * 3` becomes $\mathrm{II}(S)*3 = \mathrm{II}(\langle t : U_1 = \omega, 7 : t\rangle)*3$; notice that, since the type of $\mathrm{II}(S)$ is $\mathrm{I}(S)$, the whole program has type ω. On the other hand, product types correspond to functors, for instance `F` can be written as

$$\lambda S : \sum_{t:U_1} t.\langle s : U_1 = \mathrm{I}(S) \times \mathrm{I}(S), \langle \mathrm{II}(S), \mathrm{II}(S)\rangle : s\rangle$$

---

[5] Mihda is available at `http://jordie.di.unipi.it:8080/mihda`, where also documentation and examples are provided. A web interface to Mihda can be accesses via browser at `http://jordie.di.unipi.it:8080/pweb`.

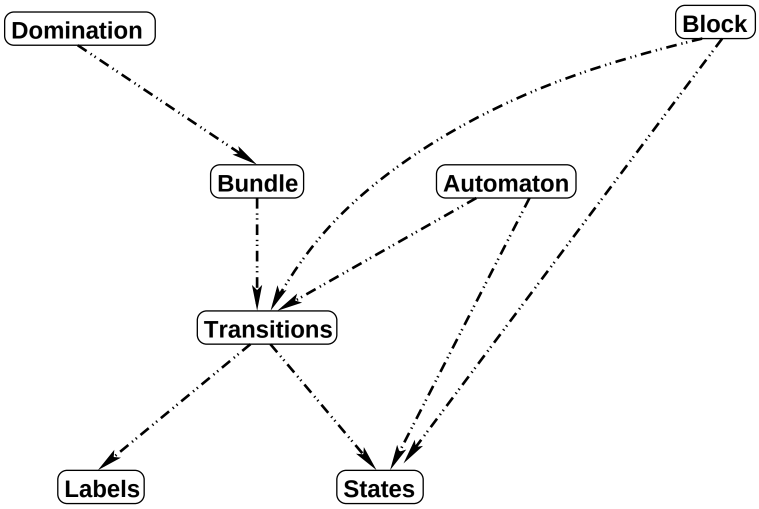


Figure 5: Mihda Software Architecture

which has type $\prod_{S:T} \sum_{s:U_1} s$, where $T = \sum_{t:U_1} t$. Even though expressive enough for our purposes, the kind of polymorphism provided by the module system of ocaml is less powerful than the polymorphism of $\lambda^{\to,\Pi,\Sigma}$; the reason is that signatures, structure and functors can be only used at "top-level" and recursion is not allowed in their definitions. The reader is referred to [13] for a deeper discussion on this topic.

Figure 4 illustrates the modules of Mihda and their dependencies. For instance, State is the module which provides all the structures for handling states and its main type defines the type of the states of the automata. Domination is the module containing the structures underlying bundle normalisation. The connections express typing relationships among the modules. For instance, since states in bundles and transitions must have the same type, then a connection exist between Bundle and Transitions modules.

The iterative construction of the minimal automaton is parameterised with respect to the modules of Figure 4. Indeed, the same algorithm can be applied to different kind of automata and bisimulation, provided that these automata match the constraints on types imposed by the software architecture. For instance, the architecture of Mihda has been exploited to provide minimisation of both HD-automata and ordinary automata (up to strong bisimilarity). Another practical benefit is that Mihda remains tightly close to its specification.

4.1 Main data structures

We describe the main data structures used in Mihda together with the properties that are relevant to show adequacy of our implementation to the coalgebraic $\lambda^{\to,\Pi,\Sigma}$ specification. Moreover, relationships between the "theoretical" objects and their Mihda counterpart is pointed out.

In the rest of the paper, we will use slanted symbols to denote names for ocaml functions and variables. A list `l` is written as $[\mathsf{e}_1; ...; \mathsf{e}_h]$ while $\mathtt{l}_i$ denotes its i-th element (i.e. `e`$_i$). Finally, we write `e` $\in$ `l` to indicate that `e` is an item of list `l`. As a general remark, notice 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 `e` $\in$ `x` such that `e` *corresponds* to e.

An automaton is made of three ingredients: Initial state, states and arrows. As far as finite state automata are concerned, it is possible to represent (finite) automata by enumerating states and transitions.

Observation 4.1 *We assume that* $1, .., n$ *are the names of a state having* n *names. 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 below we show how the classical notation for permutations is represented by* ρ*, a list of integers*

$$\left(\begin{array}{ccc} 1 & ... & n \\ i_1 & ... & i_n \end{array} \right), \qquad \rho = [i_1; ...; i_n].$$

With these notations, $[2; 1; 3]$ *represents a permutation of* 3 *elements: Namely, the permutation that exchanges* 1 *and* 2*, and leaves* 3 *unchanged.*

We adopt the notation of Observation 4.1 also to represent other functions on names. In particular, given a quadruple $\langle q, \ell, \pi, \sigma \rangle$, π is represented by means of a list of integers `pi` whose length is $|\ell|$ and whose i-th position contains $\pi(i)$ (for $i = 1, ..., |\ell|$). 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` (resp. `sigma`) *corresponds to* π (resp. σ).

HD-automata are an extension of ordinary automata, since states and labels have a richer structure carrying information on names. A state may be concretely represented 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 *group* component is its symmetry, i.e. the set of those permutations that leave the state unchanged. By the previous observation, we can represent it as a list of list of integers.

Definition 4.1 (States correspondence) *An element* State(`q`, `names`, `group`) *corresponds to a state* q *of a named set* $A = \langle Q, \lesssim, |\_|, G \rangle$ *if, and only if,*

- $q \in Q$
- $|q| =$ `length names`
- `group` *corresponds to Group.*

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

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

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

Note that type of arrows relies on type for labels. A label for π-agents is a triple whose first component is an element of L_π; 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 alternative, more simple, definition could have been obtained by embedding the `labeltype` in Arrow. Although more adherent to the definition of bundle given in Section 3.2.1, 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. Essentially, a quadruple is the transition from a state to another state. Transitions are labelled 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
```

Note that the first component of bundles is not represented. This choice is possible because implementation always deals with bundles that are obtained by applying the iterative construction $H_{i+1} = K;T_2(H_i)$. Therefore, the first component of these bundles always is S_K, the set of states of the initial automaton.

We can establish a precise connection between quadruples and objects that populate `quadtype` and, therefore, between automata and elements in `Automaton_t`.

Definition 4.2 (Quadruple correspondence) *Given a quadruple* $qd = \langle q, \ell, \pi, \sigma \rangle$, *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 4.3 (Automata correspondence) *Let* $K = \langle Q, \mathcal{T}(Q), k : Q \to \mathcal{T}(Q), \Sigma \rangle$ *be a named function representing an automaton for a* π*-agent. We say that* $(\mathtt{q}, \mathtt{qs}, \mathtt{as}, \mathtt{S})$ corresponds to K *iff,* `qs` *corresponds to* Q*; for each* $t \in k(q)$ *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* t*, and, for each* $\sigma \in \Sigma(q)$ *there is* $\mathtt{s} \in \mathtt{S}$ *such that* `s` *corresponds to* σ.

4.2 The main cycle

The step at a generic iteration of the minimisation algorithm $(H_{i+1} = K; T_2(H_i))$ can be explicitly written as follows:

$$h_{H_{i+1}}(q) = norm\ \langle T_1(\mathrm{cod}_{H_i}), \bigcup_{\langle q', \ell, \pi, \sigma \rangle \in \mathcal{S}_{\mathrm{h}_K(q)}} \{ \langle h_{H_i}(q'), \ell, \pi, \sigma'; \sigma \rangle \ \mid \ \sigma' : \Sigma_{H_i}(q') \} \rangle. \quad (9)$$

Following equation (9), we can compute $h_{H_{i+1}}(q)$ through the following steps:

(a) determine the bundle of q in the automaton, i.e. $\mathrm{h}_K(q)$;

(b) for each quadruple $\langle q', \ell, \pi, \sigma \rangle$ in $\mathrm{h}_K(q)$, apply h_{H_i} to q', the target state of the quadruple (yielding the bundle of q' in the previous iteration of the algorithm);

(c) compose all $\sigma \in \Sigma(q')$ with σ';

(d) normalise the resulting bundle.

Mihda stores the representation of the minimised automaton at the i-th iteration (i.e. h_{H_i}) in a list of *blocks* which are the most important data structures of Mihda. As said, blocks represent the equivalence classes of the kernel of each iteration and contain all those information for computing the iteration steps of the algorithm. Indeed, blocks represent both the (finite) named functions corresponding to the current iteration and its kernel. 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] (i.e. early bisimulation). The remaining fields respectively represent

- the normalised bundle with respect to the block considered as state (*norm*),
- *names* is the list of names of the bundle in *norm*,
- *group* is the group of the block,
- the functions relative to the bundle ($\underline{\Sigma}$), last field, $\underline{\Theta^{-1}}$, is the function that, given a state q, maps the names appearing in *norm* into the name of q. Basically, $\underline{\Theta^{-1}}(q)$ 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.

A graphical representation of a block is reported in Figure 6. The element `x` is the "representative state", namely it is the representative element of the equivalence class corresponding to the block. The names of the block and its group respectively are the names and the group of `x` (graphically represented by the arrow from `x` to itself in Figure 6 that aims at recording that a block also has symmetries on its names). All those states of the automaton `q` mapped on `x` are collected in the block. Function $\theta_{\mathtt{q}}$ describes "how" the block approximates the state `q` at a given iteration. Bundle *norm* of block `x` is computed by exploiting the ordering relations over names, labels and states.

A graphical representation of steps (a)-(d) above in terms of blocks is illustrated in Figure 7. Step (a) is computed by the facility `Automaton.bundle` that filters all

---

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

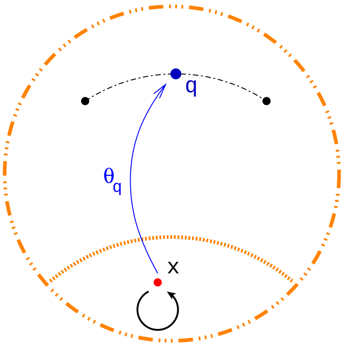
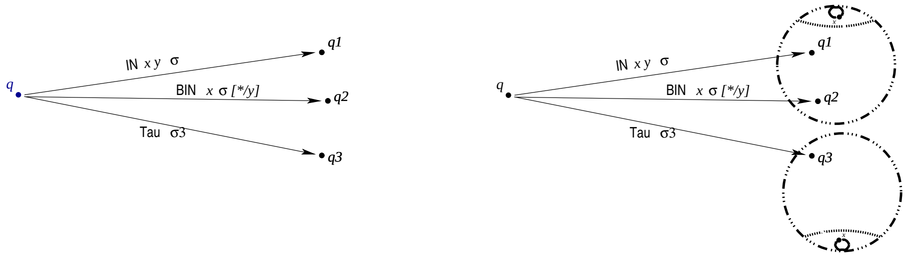
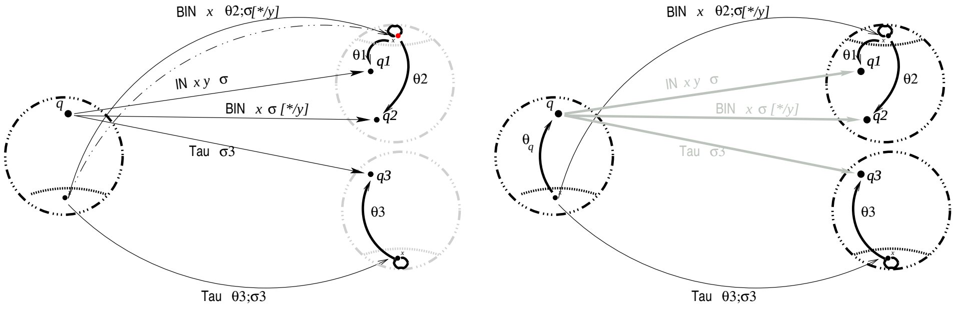


Figure 6: Graphical representation of a block



(a) Step 1

(b) Step 2



(c) Step 3

(d) Step 4

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

transitions 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 facility `Block.next` to the bundle of `q`. The operation `Block.next` substitutes all target states of the quadruples with the corresponding current block and computes the new mappings (see Figure 7(b)).
Step (c) does not seem to correctly adhere to the corresponding step of equation 9. However, if we consider that θ functions are computed at each step by composing symmetries σ's we can easily see that θ functions exactly play the rôle of σ's.
Finally, step (d) is represented in Figure 7(d) and is obtained via the function `normalise` in module `Bundle`.

The previous operations are computed by function `split`[7] that divides the states among the partitions relative to the current iterations.

```
let split blocks block =
  try
    let minimal =
      (Bundle.minimise 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.normalise
              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 minimising the reduced bundle of the first state of `block`. The choice of the state for computing `minimal` is not important: Without loss of generality, given two equivalent states `q` and `q'`, it is possible to map names of `q` into names of `q'` preserving their associated normalised bundle if, and only if, a similar map from names of `q'` into names of `q` exists.

Once `minimal` has been computed, `split` invokes `Block.split` with parameters `minimal` and `block`, while the second argument of `Block.split` is a function that computes the current normalised bundle of each state in `block` and checks whether or not it is bisimilar to `minimal`. This computation is performed by function `bisimilar` (in the module `Bisimulation`). If bisimilarity holds through $\theta_{\mathtt{q}}$ then `Some` $\theta_{\mathtt{q}}$ is returned, otherwise `None` is returned.

---

[7] We exploit two ocaml primitive functions on lists. Function *head*, *List.hd*, that takes a list and returns the first element of the list; Function *List.map*, is the usual *map* function of functional languages; given a function `f`, *List.map* `f` $[\mathtt{f(e)}_1; \ldots; \mathtt{e}_h]$ is the list $[\mathtt{f}(\mathtt{e}_1); \ldots; \mathtt{f}(\mathtt{e}_h)]$.

```
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
```

Figure 8: The main cycle of Mihda

We are now ready to comment on the main cycle of Mihda reported in Figure 8. Let `k` = (`start`, `states`, `arrows`) be an automaton. When the algorithm starts, `blocks` is the list that contains a single block collecting all the states of the automata `k`.

At each iteration, the list of blocks is splitted, as much as possible, by `split_iter` that returns a list of *buckets* which have the same fields of a block apart from the name, symmetries and the functions mapping names of destination states into names of source states. Basically, 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. 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. This condition is equivalent to say that an isomorphism can be established between `oldblocks` (that corresponds to $\ker H_i$) and `blocks` (corresponding to $\ker H_{i+1}$). 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.

5 Concluding Remarks

This paper gives a formal account of a minimisation algorithm of HD-automata that suitably represent mobile history dependent systems; the associated Mihda toolkit is also detailed.

The formal devices used in this work are coalgebras and $\lambda^{\to,\Pi,\Sigma}$, a polymorphic λ-calculus. On the one hand, coalgebras allow us to express transition systems in an elegant mathematical framework. On the other hand, $\lambda^{\to,\Pi,\Sigma}$ is used as a formal specification language that drives to a smooth implementation.

This approach has a twofold advantage. First, the coalgebraic mathematical framework accounts for the convergence proof of the minimisation algorithm on finite HD-automata. Second, using $\lambda^{\to,\Pi,\Sigma}$ as a specification language and ocaml as the implementation language of Mihda, permit to point out the tight correspondence between the specification and the implementation.

From a programming perspective, our approach enjoys a high level of modularisation. Indeed, product types and their ocaml counterpart, i.e. modules, provide the programming guidelines for adding or changing facilities that are neatly separated in different modules. For instance, Mihda can be used for minimising both HD-automata and traditional automata; or else, automata can be minimised according to different notions of equivalences. We plan to extend the Mihda toolkit with facilities to handle other notions of equivalences (e.g. open bisimilarity) and other foundational calculi for global computing (e.g. the asynchronous π-calculus, the fusion calculus).

Some experimental results of Mihda can be found in [5] and seem quite promising. The π-calculus specification of the Handover Protocol (borrowed from [25, 16]) has been minimised running Mihda on a machine equipped with an AMD Athlon$^{\text{TM}}$XP 1800+ dual processor with 1Giga RAM. The time required for minimising the automata is very contained (few seconds). The size of the minimal automata in terms of states and transitions is sensibly smaller than their non-minimised version. (the number of states and transitions in the minimal automaton are reduced of a factor 7). In the future, we plan to improve efficiency incorporating supports for symbolic approaches based on Binary Decision Diagrams.

As a final comment, we remark that different proofs for the convergence theorem (Theorem 3.2, page 27), relying on well-known results in coalgebras (e.g. [20, 26]) can be given. However, the proof given in this paper has two advantages: First, it is conceptually more simple and, second, it is based on those constructions used in the implementation thus providing hints for correctness of Mihda.

References

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[2] Edmund M. Clarke and Jeanette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

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

[4] 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.

[5] Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto. From co-algebraic specifications to implementation: The Mihda toolkit. In *Second International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, November 2003. To appear.

[6] Riccardo Focardi and Roberto Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1), 1995.

[7] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In Giuseppe Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[8] 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.

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

[10] 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.

[11] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[13] John C. Mitchell. *Foundations for Programming Languages.* MIT press, 1996.

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

[15] Ugo Montanari and Marco Pistore. π-calculus, structured coalgebras, and minimal HD-automata. In Mogens Nielsen and Branislav Roman, editors, *MFCS: Symposium on Mathematical Foundations of Computer Science*, volume 1983 of *LNCS*. Springer Verlag, 2000. An extended version will be published on Theoretical Computer Science.

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

[17] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

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

[19] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In Roland Backhouse and José N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[20] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.

[21] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory.* Prentice-Hall, New York, 1988.

[22] Davide Sangiorgi. A theory of bisimulation for the pi-calculus. *Lecture Notes in Computer Science*, 715, 1993.

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

[24] Jeffrey D. Ullman. *Elements of ML Programming: (2nd ed).* Prentice_Hall, ml97 edition, December 1997.

[25] Bjorn Victor and Faron Moller. The mobility workbench: A tool for the pi-calculus. *Lecture Notes in Computer Science*, 818:428–??, 1994.

[26] James Worell. Terminal sequences for accessible endofunctors. In Lecture Notes in Computer Science, editor, *Category Theory and Computer Science*, volume 19, 1999.