Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 331 (2005) 325–365
Theoretical Computer Science
www.elsevier.com/locate/tcs

Coalgebraic minimization of HD-automata for the π-calculus using polymorphic types☆

Gianluigi Ferrari, Ugo Montanari*, Emilio Tuosto

Dipartimento di Informatica, Via F. Buonarroti 2, 56127 Pisa, Italy

Abstract

We introduce finite-state verification techniques for the π-calculus whose design and correctness are justified coalgebraically. In particular, we formally specify and implement a minimization algorithm for HD-automata derived from π-calculus agents. The algorithm is a generalization 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.

Keywords: Name passing calculi; Finite state verification; Partition refinement algorithm; Bisimulation checking; Dependent types; Co-algebras

1. Introduction

One of the main advantages of applying formal methods to system design is the possibility of constructing an abstraction of systems and their computations that are, at least at a certain extent, amenable of automatic verification. Several process-algebraic techniques have been developed for reasoning about concurrent and distributed systems. For instance, it is possible to verify whether an implementation is "coherent" with its specification by checking a suitable behavioural equivalence among them. Another example is the *information leak* detection; in [7] the analysis of information flow is done by modelling the system as a

☆ This work has been supported by EU-FET project **PROFUNDIS** IST-2001-33100.

* Corresponding author.
E-mail addresses: giangi@di.unipi.it (G. Ferrari), ugo@di.unipi.it (U. Montanari), etuosto@di.unipi.it (E. Tuosto).

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.09.021

CCS-process P and then verifying that it is equivalent to 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.

Finite state automata (e.g., labelled transition systems) provide a foundational model underlying effective verification techniques of process-algebraic theories 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) 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 [3,9,20]. The algorithm yields the minimal realization of the initial automaton by "grouping" all the equivalent states in a single state. Moreover, it is often convenient, from a computational point of view, to verify properties on the minimal automaton rather than on the original one. Indeed, minimization 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.

Global computing, i.e., networks of stationary and mobile components, are becoming the prominent example of large-scale distributed systems. 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 centralized 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. As pointed out by Milner [12] a great challenge is to "develop calculi, theories and automated tools that allows descriptive and predicative analysis of global computing systems at each level of abstraction".

Name passing calculi (e.g., the π-calculus [11,13,26]) probably are the best known and acknowledged models which provide a rich set of techniques for reasoning about global computing systems. *History-dependent automata* (HD-automata for short) have been proposed in [4,17,18,21] as a new effective automata-based model for name passing calculi. 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.

Depending on the level of abstraction, different definitions of HD-automata have been provided. They have been characterized as automata over permutation algebras, whose ingredients are sets of names and groups of permutations (renaming substitutions) on them. This foundational 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 [8,18,22]. At a more concrete level, HD-automata have been introduced by exploiting the notions of *named sets* and *named functions*; elements of a named set are equipped with names that are defined up to specific groups of name permutations called *symmetries*. HD-automata are defined as coalgebra on a category whose objects are named sets and whose arrows are named functions. The two definitions above are, at the same time, equivalent [18] and complementary; indeed, 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 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 minimization of (finite state) HD-automata according to the second definition. The algorithm is a generalization of the partition refinement algorithm for minimizing ordinary automata (up to bisimilarity) [9].

Coalgebraic specifications have been proved very useful to formally describe the behaviour of process calculi. However, the development of effective verification techniques based on coalgebraic foundations has had more limited success. The present paper intends to explore this issue. In particular, we will address the following question:

Can we define effective verification techniques for name passing calculi which can be justified coalgebraically?

The main results of this paper are

(1) a coalgebraic theory for the π-calculus,
(2) a minimization algorithm whose design and correctness are justified coalgebraically.

In particular, we illustrate the features of the framework in the development of a toolkit, called Mihda, providing facilities to minimize labelled transition systems for name passing calculi.

A distinguished feature of our approach is the exploitation of a polymorphic λ-calculus, $\lambda^{\rightarrow,\Pi,\Sigma}$ [16], for describing data and control components of the minimization procedure. The type system of $\lambda^{\rightarrow,\Pi,\Sigma}$ encompasses polymorphic and dependent types. We exploit polymorphism for abstracting from unimportant features (with respect to the minimization 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 $\lambda^{\rightarrow,\Pi,\Sigma}$ is also an effective basis:

- to drive the implementation choices; for instance, the specification naturally suggests an ML-like language (we chose ocaml) since the type system of $\lambda^{\rightarrow,\Pi,\Sigma}$ is a generalization of the ML-type system.
- to show the correctness of the implementation.

A pure set-theoretic presentation of HD-automata would have work as well (see the interpretation of $\lambda^{\rightarrow,\Pi,\Sigma}$ constructs in Section 2.2). However, in this case we would have two main drawbacks. First, the sets corresponding to the models of the types should explicitly appear in all constructions instead of the more compact-type expressions. Second, the connection between the coalgebraic framework and the implementation Mihda would no longer be explicit. Indeed, Mihda builds directly on the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification: the ML-types of the implementation are in one-to-one correspondence with the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification. In other words, the set-based description would have been heavier and the correctness of Mihda would have been more obscure than in the $\lambda^{\rightarrow,\Pi,\Sigma}$ presentation.

Structure of the paper: Section 2 collects the formal ingredients our work bases 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 $\lambda^{\rightarrow,\Pi,\Sigma}$. 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 $\lambda^{\rightarrow,\Pi,\Sigma}$ 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 minimization algorithm is given in Section 3.3 where also its convergence on finite HD-automata is proved.

Section 4 shows the correspondence between the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification of the minimization algorithm and Mihda, its actual implementation in ocaml.

2. Preliminaries

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

2.1. Coalgebras

Coalgebras provides a very elegant mathematical machinery to describe the behaviour of process calculi. This section reviews some elementary notions of coalgebras. In particular, we will restrict our attention on coalgebras over sets and functions.

Definition 2.1 (*Functor*). Let $\mathcal{C}$ be a category; an (*endo-*)*functor* $\mathcal{F}$ over $\mathcal{C}$ maps objects to objects and arrows to arrows as follows:

- for each arrow $f : A \to B$, $\mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(B)$;
- for each object A, $\mathcal{F}(\mathrm{id}_A) = \mathrm{id}_{\mathcal{F}(A)}$;
- for all arrows $f : A \to B$ and $g : B \to C$, $\mathcal{F}(f; g) = \mathcal{F}(f); \mathcal{F}(g)$.

Fig. 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

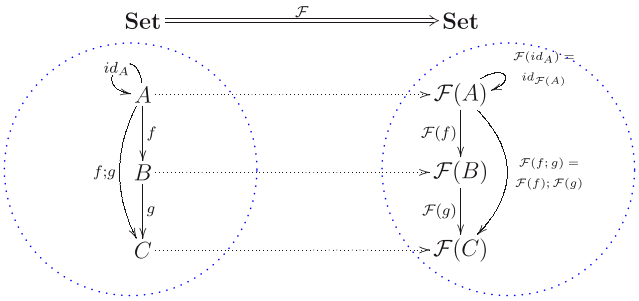


Fig. 1. Functor over *Set*.

$A \mapsto \wp(A)$, 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

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

Then, by definition,

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

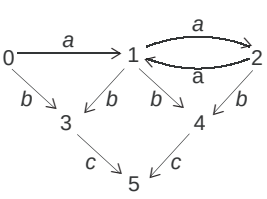
This proves that the powerset operation is functorial.

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

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

A $\mathcal{F}$-coalgebra is a pair $\langle A, \alpha : A \to \mathcal{F}(A)\rangle$ where α 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.



$\alpha(0) = \{\langle a,1\rangle, \langle b,4\rangle\}$
$\alpha(1) = \{\langle a,2\rangle, \langle b,3\rangle, \langle b,4\rangle\}$
$\alpha(2) = \{\langle a,1\rangle, \langle b,4\rangle\}$
$\alpha(3) = \{\langle c,5\rangle\}$
$\alpha(4) = \{\langle c,5\rangle\}$
$\alpha(5) = \phi$

Notice how, for each state $q \in \{0, ..., 5\}$, $\alpha(q)$ yields all the immediate successor states of q and the corresponding labels. Nonetheless, the coalgebraic theory we developed is sufficient to effectively address verification issues. Indeed, $(\ell, q') \in \alpha(q)$ if, and only if, $q \xrightarrow{\ell} q'$.

A $\mathcal{F}$-coalgebra (A, α) is *final* provided that for any $\mathcal{F}$-coalgebra (B, β) there exists precisely one coalgebra morphism $f : (B, \beta) \to (A, \alpha)$. 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 show that the powerset functor $\wp(\_)$ does not admit final coalgebra. For many functors over sets, however, the final coalgebra exists. It is well known that for continuous functors the final coalgebra is obtained as the limit of the terminal 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 [23].

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

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

has a final coalgebra. The powerset functor $\wp_{\text{fin}}(\_)$ is the standard example of *bounded* functor. It has been proved that bounded functors admit final coalgebras [23].

Throughout this paper, we will exploit standard coalgebraic techniques to define HD-automata and their finite state verification techniques. In particular, the iteration of the functor along the terminal sequence will converge in a finite number of steps and will construct the minimal HD-automaton when applied to a finite HD-automaton. Since all our coalgebraic constructions live in the category of finite (named) sets we will not construct the final coalgebra: The image of the functor along the terminal sequence is not final in such a category because it still is a finite set. Nonetheless, the coalgebraic theory we developed is sufficient to effectively address the issues related to the design of verification techniques. Indeed, the iteration of the functor along the terminal sequence provides a declarative specification of the minimization algorithm and the formal machinery to prove its termination is justified coalgebraically.

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

This section reviews some basic type-theoretic notions underlying the description of languages which support the organization of applications into autonomous (compilable) modules exploiting explicit-type information (e.g., the ML module language). We refer to [16] 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 $t \to t'$ 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.

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 $\sum_{x:t} t'$. 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.

General sum types encompass tuple types, indeed, provided that x does not occur free in t', $\sum_{x:t} t'$ is equivalent to $t \times t'$. In these cases, we will sometime write $t \times t'$ instead of $\sum_{x:t} t'$.

We now introduce a summary of the $\lambda^{\rightarrow,\Pi,\Sigma}$ predicative calculus with products and sums as reported in [16]. 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 M of $\lambda^{\rightarrow,\Pi,\Sigma}$ is given by

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

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. We will use τ to denote types.

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 describe 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 notation of general product types is reminiscent of the standard notation for the cartesian products over a family of sets indexed by an index set A:

$$\prod_{a\in A} B_a = \left\{ f : A \to \bigcup_{a\in A} B_a \mid \forall a \in A. f(a) \in B_a \right\}, \tag{1}$$

where, the elements of this type are functions f such that $f(a) : t'[a/x]$, for each $a : t$.

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

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

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

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

Similarly to general products, the type of general sums correspond to the disjoint union of sets

$$\sum_{a\in A} B_a = \{\langle a, b\rangle \mid a \in A \wedge b \in B_a\}. \tag{2}$$

General products and sums of $\lambda^{\to,\Pi,\Sigma}$ can be set-theoretically interpreted exactly as in (1) and (2). For instance *Henkin models* (see [16, Chapter 9]) interpret each type as a set and, given the polymorphic type $\tau = \prod_{x:\tau_1} \tau_2$, if τ_1 is interpreted as A, the interpretation of τ is (1), namely, the elements that inhabit τ are functions from A to $\bigcup_{a\in A} B_a$ (that is the interpretation of $\tau_2[\tau_1/x]$) such that $f(a) \in B_a$, for any $a \in A$.

2.3. The π-calculus

This section briefly summaries syntax and semantics of the π-calculus [13]. We refer to [10,26] for more detailed presentations.

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

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

where $r(A)$ is the rank of the *process identifier* A. The occurrences of y in $x(y).P$ and $\nu y P$ are bound; *free names* are defined as usual and $\mathrm{fn}(P)$ indicates the set of free names of agent P. We assume that, for each identifier A, there is a definition $A(y_1, \ldots, y_{r(A)}) \stackrel{\mathrm{def}}{=} P_A$ (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).

Table 1
Early operational semantics

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

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

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

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

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

STRUCT $\dfrac{P \equiv P' \xrightarrow{\mu} Q' \equiv Q}{P \xrightarrow{\mu} Q}$

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

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

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 operational 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 and 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
- $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$, if $x \notin \mathrm{fn}(P)$.

Several bisimulation equivalences have been introduced for the π-calculus [26]; they are based on direct comparison of the observable actions π-agents can perform. They can be strong or weak, early, late [13] or open [25]. 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) [21].

Definition 2.3 (*Early bisimulation*). A binary relation over a set of agents $\mathcal{B}$ 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 $P \sim Q$, if there exists a bisimulation $\mathcal{B}$ such that $P\ \mathcal{B}\ Q$.

3. A minimization procedure for HD-automata

This section introduces the formal definitions for types and operations exploited in the minimization algorithm on HD-automata for π-agents. Our approach consists of formally describing the data and the control components of the minimization procedure as $\lambda^{\rightarrow,\Pi,\Sigma}$ expressions. This provides us with some benefits. First, it enables us to formally prove termination of the minimization algorithm. Second, the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification has to be considered as an intermediate step toward the actual implementation, Mihda. In Section 4, we will show the strict correspondence between the $\lambda^{\rightarrow,\Pi,\Sigma}$ specification and Mihda.

Since our construction consists of several interrelated components the types of $\lambda^{\rightarrow,\Pi,\Sigma}$ provide an effective mechanism to deal with and control the dependencies among the various components. Moreover, the formal specification of each component is rather compact and self contained. The set-theoretic presentation of HD-automata has been given in previous works [18,21]. The set-theoretic presentations can be viewed as a "macro expansion" of $\lambda^{\rightarrow,\Pi,\Sigma}$ types (see Example 3.1). Indeed, all the types introduced in this paper have set-theoretic models due to the fact that we stick to the finite case.

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

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

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

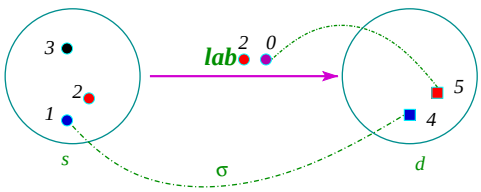


Fig. 2. A HD-automaton transition.

respectively. The transition is labelled by lab and exposes two names: Name 2 of s and a fresh name 0. Notice that name 3 of s is "deallocated" along such transition.

It is worth to emphasize that name creation is simply handled by associating in the target state a name not in the source state, while a name in a state s can be deallocated when it is not involved in any transition from s.

Remark 3.1. In order to avoid cumbersome details regarding the definitions of $\lambda^{\rightarrow,\Pi,\Sigma}$ types, we make some simplifying assumptions. First, we assume as given the primitive types (e.g., boolean, integers, strings, etc.) 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

This section introduces the types of named sets, named functions together with their main features. We describe HD-automata and their minimization algorithm as a coalgebra over a functor defined on the category of named sets. Such category has named sets as objects and named functions as morphisms. Here, $\lambda^{\rightarrow,\Pi,\Sigma}$ types will be exploited for specifying both objects and morphisms of the category of named sets. Clearly, because of the set-theoretic interpretation of $\lambda^{\rightarrow,\Pi,\Sigma}$, the category of named sets is a subcategory of *Set*. Moreover, since all our constructions lives in the category of finite named sets, the minimal automaton can be represented by simply exploiting non recursive types. Indeed, polymorphism allows us to pass to the functor a different type at each iteration obtained by applying the finite powerset functor; however, all such types can be casted to the type for named sets because of the finiteness constraint.

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 [18]) 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

$$\mathsf{N} \stackrel{\triangle}{=} \prod_{n:\omega} 1 \cdots n.$$

For instance $\mathsf{N}(4)$ is the interval of the natural numbers from 1 to 4. By convention, type $\mathsf{N}(0)$ is interpreted as the empty set. 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.

A *permutation algebra* is an algebra whose operations are *finite kernel* [1] permutations of names. In a permutation algebra, permutations are considered as operations that transform the elements of the support. In [18] a permutation algebra for π-calculus has been introduced; the support of the algebra is the set of π-agents where $\rho(P)$ is interpreted as $P\rho$, namely

[1] A permutation of names is a bijective functions ρ on the set of names $\mathcal{N}$. A *finite kernel* permutation is a permutation ρ such that $\rho(n) \neq n$ for finitely many names.

the application of substitution ρ to the agent P. In this context $\mathrm{sym}(P)$, the symmetry of P, is defined to be $\mathrm{sym}(P) \stackrel{\triangle}{=} \{\rho | P\rho = \rho\}$ (notice that $\mathrm{sym}(P)$ is a group of permutations).

Named sets represent states of HD-automata.

Definition 3.1 (*Named sets*). The type of named sets is

$$\mathtt{NS} \stackrel{\triangle}{=} \sum_{Q:\mathtt{U}_1} \sum_{\lesssim:Q\times Q\to\mathtt{bool}} \sum_{|\_|:Q\to\omega} \prod_{q:Q} \wp_{\mathrm{fin}}(\mathtt{N}(|q|) \to \mathtt{N}(|q|)).$$

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 generalized 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 states; function $|\_|_A$ is called the weight function (of A) and associates the number of names to elements in Q_A; the generalized product type of the last component assigns a set of permutations of names of q, namely, the symmetry of q. In the following we write $q \lesssim_A q'$ ($q \not\lesssim_A q'$) instead of writing $\lesssim_A(q, q') = \mathtt{true}$ ($\lesssim_A(q, q') = \mathtt{false}$).

Example 3.1. Let us consider the π-calculus agent $A(x, y)$ given by

$$A(x, y) \stackrel{\triangle}{=} (\nu z)(\bar{x}z.P + \bar{y}z.P).$$

The state q_A that represents $A(x, y)$ has two local names 1 and 2 (namely, $|q_A| = 2$); the symmetry of q_A is the set containing the identity permutations (of names 1 and 2) and the permutation that exchanges 1 with 2.

The type NS is the finite counterpart of permutation algebras. In other words, we do not consider permutations as bijections of the whole set of names, but only as bijections of the "relevant" names of a state that, according to Definition 3.1, are finite. Indeed, notice that $\mathcal{G}_A(q)$ plays the role of a symmetry and is a list of permutations of $\mathtt{N}(|q|)$, the names of q, (the natural numbers in the interval $1, \ldots, |q|$).

Lemma 3.1. *Let* $\mathbf{0}$ *be the empty substitution* (*i.e., the substitution whose domain is the empty set of names*), *then* $\mathcal{G}_A(q) = \{\mathbf{0}\} \iff |q| = 0$.

Proof. The proof follows by Definition 3.1 and by the fact that if $|q| = n > 0$ and id_n is the identity on $\mathtt{N}(n)$ then $\mathrm{id}(q) = q$. □

Hereafter, we often have to compose functions with sets of functions, hence we adopt the following notation. Given a set of functions F and a function g such that it can be composed

[2] Formally, A should be written as $\langle Q_A, \langle \lesssim_A, \langle |\_|_A, \mathcal{G}_A\rangle\rangle\rangle$. When it is clear from the context, we adopt the compact notation $\langle Q_A, \lesssim_A, |\_|_A, \mathcal{G}_A\rangle$ that avoids writing many brackets. The same notational abuse is adopted throughout this paper.

with all functions in F, we let $F; g$ to denote the set of functions given by $\{f; g | f \in F\}$ (symmetrically for $g; F$). Similarly, if G is a set of functions that can be composed with all functions in F, then $F; G = \{f; g | f \in F \text{ and } g \in G\}$.

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

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

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

Given a named function $H : \mathsf{NF}$, we use the following shorthands:

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

which correspond to the projections of the sum type NF.

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

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

The type of named functions is a generalized 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)–(3) naturally emerges from the interpretation of named function H as a coalgebraic description of a HD-automaton: Elements in $h_H(q)$ 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. Symmetries of $h_H(q)$ are those name permutations that when applied do not change $h_H(q)$, the set of transition from q. Condition (1) states that any permutation belongs to the symmetries of $h_H(q)$ 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 $\Sigma_H(q)$. 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 in $|q|$ has a unique "meaning" along transitions in $h_H(q)$.

Remark 3.2. If $h_H(q)$ has no name (i.e., $|h_H(q)| = 0$) then $\mathcal{G}_A(h_H(q))$ is the singleton $\{\mathbf{0}\}$ that in turn implies that *any* $\sigma \in \Sigma_H(q)$ is the empty substitution $\mathbf{0}$.

An ordinary function f on sets induces a partition on its domain. The relation $=_f \subseteq \text{dom}_f \times \text{dom}_f$ given 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 $=_f$, namely $\text{dom}_f/_{=_f}$. Let f and g be two functions such that $\text{dom}_f = \text{dom}_g$, we can define $f \simeq g \iff \ker f = \ker g$. Relation $\simeq$ 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 $\ker h_H$;
- given $A, B \in \ker h_H$, $A \lesssim B$ if, and only if, for some $a \in A$ and some $b \in B$, $h_H(a) \leq_{\mathrm{cod}_H} h_H(b)$;
- the weight of an element $A \in \ker h_H$ is $|h_H(a)|_{\mathrm{cod}_H}$, for $a \in A$;
- the group of $A \in \ker h_H$ is $\mathcal{G}_{\mathrm{cod}_H}(h_H(a))$, for $a \in A$.

The named set $\ker H$ 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.

Definition 3.4 (*Composition of named functions*). Let $H, K : \mathtt{NF}$ be two named functions. We say that H and K can be *composed* 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 $\mathrm{dom}_{H;K} = \mathrm{dom}_H$, $\mathrm{cod}_{H;K} = \mathrm{cod}_K$, $h_{H;K} = h_H; h_K$ and $\Sigma_{H;K} = \lambda q \in \mathrm{dom}_{H;K}.\Sigma_K(h_H(q)); \Sigma_H(q)$.

Proposition 3.1. *Let* H *and* K *be two named functions that can be composed. Then,* $H; K$ *is a named function.*

Proof. Let us consider $q \in \mathrm{dom}_{H;K}$ and $\sigma \in \Sigma_{H;K}(q)$. First, recall that $\Sigma_{H;K}(q) = \Sigma_K(h_H(q)); \Sigma_H(q)$, therefore, there are $\sigma_1 \in \Sigma_K(h_H(q))$ and $\sigma_2 \in \Sigma_H(q)$, such that $\sigma = \sigma_1; \sigma_2$. We have to show that the conditions (1)–(3) hold.

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

$$
\begin{array}{ll}
\mathcal{G}_{\mathrm{cod}_{H;K}}(h_{H;K}(q)); \sigma & \\
= & \text{Definition 3.4 and } \sigma = \sigma_1; \sigma_2 \\
\mathcal{G}_{\mathrm{cod}_K}(h_K(h_H(q))); (\sigma_1; \sigma_2) & \\
= & \text{associativity of composition} \\
(\mathcal{G}_{\mathrm{cod}_K}(h_K(h_H(q))); \sigma_1); \sigma_2 & \\
= & \text{Definition 3.2 (condition 1 on } K) \\
\Sigma_K(h_H(q)); \sigma_2 & \\
\subseteq & \text{def. of } \Sigma_{H;K} \\
\Sigma_{H;K}(q). &
\end{array}
$$

For the reverse inclusion, we must prove that σ can be written as composition of a permutation in $\mathcal{G}_{\mathrm{cod}_{H;K}}(h_{H;K}(q))$ and a $\sigma' \in \Sigma_{H;K}(q)$. By Definition 3.2, $\sigma_1 \in \mathcal{G}_{\mathrm{cod}_K}(h_K(h_H(q)))$; σ'_1, for a suitable $\sigma'_1 \in \Sigma_K(h_H(q))$; 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 3.4, $\sigma; \mathcal{G}_{\mathrm{dom}_H}(q) \subseteq \Sigma_K(h_H(q)); \Sigma_H(q)$:

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

that completes the proof.

Condition 3. This trivially holds because injectivity is preserved by composition and, by definition of named function, both $\Sigma_K(h_H(q))$ and $\Sigma_H(q)$ contains only injective functions and, for any q, $\Sigma_{H;K}(q) = \Sigma_K(h_H(q)); \Sigma_H(q)$, by Definition 3.4. □

We conclude this section by emphasizing that named functions will provide the formal mean to describe a generic step of the iterative minimization 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 will be exploited for specifying *blocks*, the main data structure of Mihda. Basically, a block collects those states considered equivalent at a generic iteration. Hence, the block intuitively *corresponds* to an element of the partition induced by the kernel of the named function associated to a generic step of the iterative algorithm.

3.2. HD-automata for π-calculus

We now present the coalgebraic specification of HD-automata for the early semantics of the π-calculus. Even if our constructions are tailored to the π-calculus they can be extended to handle name passing calculi in general.

3.2.1. Bundles for the π-calculus

We represent π-calculus labels through the enumeration type L given by

$$\mathsf{L} \stackrel{\triangle}{=} \texttt{TAU}, \texttt{IN}, \texttt{OUT}, \texttt{BOUT}, \texttt{BIN}.$$

We assume that the position of the elements of L gives the ordering relation. Let $|\_|$ be the weight map associating to each π-label l a set having as many indexes as are the names l refers to. The weight map is defined as follows:

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

No name is associated to the synchronization 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. All but label BIN have a corresponding label in the transition system of π-calculus illustrated in Section 2.3. Transitions labelled by BIN correspond to π-calculus input transitions whose object name is fresh, namely, the object name does not appear in the free names of the agent performing the transition. As usual in the π-calculus literature, 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 $x\,y.P$ or $(v\,y)\bar{x}\,y.P$, 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 names occurring in a transition are related to names of source and target states. Therefore, we have the following definitions.

The type qd of *quadruples* is given by

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

Let $D : \mathsf{NS}$ be a named set and $t : \mathsf{qd}(D)$ be a quadruple on D, then to enhance readability we will adopt the following shorthands:

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

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 map indexes of ℓ_t and names of τ_t to suitable names (of the source state of the transition), respectively. Both π_t and σ_t are needed to establish a relationship between indexes in labels and names local to states or between names of different states.

Example 3.2 (*Transitions of HD-automata*). Let $A(x, y) \triangleq (v\,z)(\bar{x}z.P + \bar{y}z.P)$ the π-agents introduced in Example 3.1 the transitions from the state $q_{A(x,y)}$ corresponding to $A(x, y)$ are described by the quadruples

$$t_1 = \langle q_P, \ell_1, \pi_1, \sigma_1\rangle, \quad t_2 = \langle q_P, \ell_2, \pi_2, \sigma_2\rangle$$

where q_P is the state corresponding to P, $\ell_1 = \ell_2 = \mathtt{BOUT}\ \pi_1 : 1 \mapsto x$ and $\pi_2 : 1 \mapsto y$. Moreover, assuming that $z \in \mathrm{fn}(P)$, both σ_1 and σ_2 map z to the fresh name 0. Finally, if $\mathrm{fn}(P) = \emptyset$ then both σ_1 and σ_2 would be the empty substitution $\mathbf{0}$ and the symmetry of q_P would have been the singleton containing $\mathbf{0}$.

Remark 3.3. In the case of BOUT and BIN labels have to deal with name generation events. The information about new names is given in σ_t. As shown in Example 3.3, if t is a transition which refers to a bound name, then σ_t maps the bound name (of t) to 0. In this respect, 0 should be properly considered as a placeholder that signals name generation events, namely that a name of the target state has been generated during the transition. No name is mapped to 0 by σ_t when ℓ_t is TAU, IN or OUT.

Definition 3.5 ($\trianglelefteq$). Given two totally ordered sets $(A_1, \lesssim_1)$ and $(A_2, \lesssim_2)$, we define $\trianglelefteq$ to 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 $\forall q' \in A_1.q' \lesssim_1 q \Rightarrow f(q') = g(q') \wedge f(q) \lesssim_2 g(q)$.

Proposition 3.2. *Relation $\trianglelefteq$ is a partial order.*

Quadruples can be totally ordered. We have that $t \sqsubseteq t'$ if, and only if,

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

The intuition is that $\sqsubseteq$ is a lexicographic ordering obtained by taking advantage of the ordering relations on the quadruple components. The relevance of $\sqsubseteq$ will emerge later to define the action of the functor over HD-automata for π-agents.

Proposition 3.3. *Relation $\sqsubseteq$ is a total order.*

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

$$\mathsf{B} \triangleq \sum_{D:\mathsf{NS}} \wp_{\mathrm{fin}}(\mathsf{qd}(D)).$$

A bundle over a named set D is a pair $\langle D, S\rangle$ where D is a named set and S is a finite set of quadruples on D. As usual, we assign names to the components of a bundle $\beta : \mathsf{B}$; 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 the total ordering $\sqsubseteq$ to bundles (over the same support).

Definition 3.6 (*Bundles' ordering*). Let $\beta_1, \beta_2 : \mathsf{B}$ 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 $\langle D_{\beta_1}, \mathcal{S}_{\beta_1} \backslash \{t_1\}\rangle \precsim \langle D_{\beta_1}, \mathcal{S}_{\beta_2} \backslash \{t_2\}\rangle$.

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

Proposition 3.4. *Relation $\precsim$ is a total order.*

3.2.2. Auxiliary operations

We now show how bundles can be *casted* to named sets. Since Definition 3.6 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, namely:

$$\{\!|\_|\!\} \triangleq \lambda\beta : \mathsf{B}. \bigcup_{t \in \mathcal{S}_\beta} \bigcup_{\substack{n \,:\, |\ell_t| \\ m \,:\, \mathsf{N}(|\tau_t|_D)}} \{\pi_t(n), \sigma_t(m)\} \backslash \{0\};$$

the weight function on bundles is the cardinality of $\{\!|\beta|\!\}$ (for any bundle β) and is denoted by $\lfloor\beta\rfloor$.

In the minimization algorithm, bundles play the role of states along the iterations of the algorithm. Hence, the names of a bundle obey the same constraints of names of states and, according to Remark 3.3, the natural number 0 should not be considered a name of the bundle.

Let β be a bundle and ρ be a permutation of its names (ρ permutes $\{\!|\,\beta\,|\!\}$). Then $\beta\rho$ denotes the bundle whose support is D_β and whose step is given by

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

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

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

The most important operation on bundles is *normalization*. This operation is needed 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 received name is inactive in the source of the transition.

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

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

are bisimilar only if, for any name substituted for u, their continuations remain bisimilar. However, P has a free input transition which corresponds to choice of name z in the early semantics, while Q has not. Thus, unless this transition is recognized 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 considered. Being z inactive in P, choosing name z is like choosing 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.

Fig. 3 illustrates the HD-automata corresponding to P and Q (the formal construction will be presented in Section 3.2.4).[4] Transition with label `IN` x z (drawn with dashed lines)

[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.

[4] The convention adopted for names in the π-component of transitions in Fig. 3 is that the name in the ith position is $\pi(i)$. For instance, sequence $x\ y$ stands for $\pi(1) = x$ and $\pi(2) = y$.

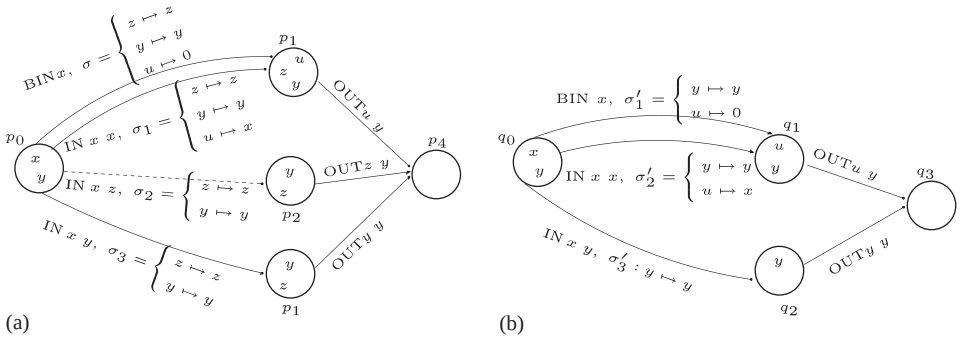


Fig. 3. Redundant transitions: (a) HD-automaton for P and (b) HD-automaton for Q.

is redundant in the automata of P. States p_0 and q_0 in Fig. 3 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 role 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 the automaton obtained by removing the redundant input transition which is exactly the automaton for Q.

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

The normalization of a bundle is done in different steps. First, the bundle is reduced by removing all the possibly redundant input transitions by applying the function *reduce*:
$\mathrm{reduce} \triangleq \lambda\beta : \mathsf{B}.\langle D_\beta,\ \ \mathcal{S}_\beta \backslash \{t \in \mathcal{S}_\beta | \mathit{dominated}(\beta)(t)\}\rangle$
where, *dominated* expresses the condition for quadruples' redundancy (for the early semantics) and it is defined as follows:

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

The underlying intuition is that a transition t is dominated in 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.

Notice that not all dominated transition are redundant transitions. Dominated transitions are only used to compute the *active names of a bundle*. The active names are those names

of the reduced bundle defined by:

$$\text{an} \stackrel{\triangle}{=} \lambda\beta : \mathsf{B}.\{\!|\, \text{reduce}(\beta)\, |\!\}.$$

We want to point out that the concept of active names of a bundle is different from the concept of active names of a process. Active names of a bundle do not involve any notion of "future" behaviour; they are characterized 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:

$$\text{rem\_in} \stackrel{\triangle}{=} \lambda\beta : \mathsf{B}.\lambda N : \wp_{\text{fin}}(\omega).\mathcal{S}_\beta \backslash \{t \in \mathcal{S}_\beta | \ell_t = \mathtt{IN} \wedge \pi_t(2) \notin N\}.$$

Finally, the normalization of a bundle is obtained by applying the function norm.

$$\begin{aligned}\text{norm} \stackrel{\triangle}{=}\ & \lambda\beta : \mathsf{B}.\\ & \quad \text{let } \beta' = \text{rem\_in } \beta\ (\text{an}(\beta))\\ & \quad in\ \ \min_{\sqsubseteq}(\beta'\theta \mid \theta : \text{an}(\beta) \to \mathsf{N}(|\text{an}(\beta)|) \text{ is a bijective substitution}).\end{aligned}$$

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

Observation 3.1. *The definition of norm requires existence of a minimal* (*representative*) *bundle. Recalling that, for any bundle* β, $\mathcal{S}_\beta$ *is finite* (*by definition*) *and the set of names of* β *is finite, we conclude that the set*

$$\{\beta'\theta \mid \theta : \text{an}(\beta) \to \mathsf{N}(|\text{an}(\beta)|)\ \textit{is a bijective substitution}\}$$

is finite as well, therefore, the minimal element exists since $\sqsubseteq$ *is a total order*.

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 transitions are collected in a bundle β' and the substitution which makes β' minimal is applied to it.

We have the following results.

Lemma 3.2. *If* $\beta, \beta' : \mathsf{B}$ *are such that* $\mathcal{S}_\beta \subseteq \mathcal{S}_{\beta'}$, *and let* $\beta_1 = \text{norm } \beta$ *and* $\beta'_1 = \text{norm } \beta'$ *then* $\mathcal{S}_{\beta_1}; \theta_\beta^{-1} \subseteq \mathcal{S}_{\beta'_1}; \theta_{\beta'}^{-1}$.

Proof. The thesis easily follows from the definition of application of a substitution to a bundle. □

Lemma 3.3. *For each* $\beta : \mathsf{B}$, $\mathcal{S}_{\text{norm } \beta} \subseteq \mathcal{S}_\beta; \theta_\beta^{-1}$.

Proof. By construction, norm β is the bundle obtained by removing dominated quadruples from β and by renaming its names through θ_β^{-1}. □

$$
\begin{array}{l}
T_1 \stackrel{\triangle}{=} \lambda A : \mathtt{NS}. \\
\quad \langle\ B = Q_A \times \wp_{\mathrm{fin}}(\mathtt{qd}(Q_A)), \\
\quad\ \lesssim\ = \lambda(\beta, \beta') : B \times B. \\
\quad\ \textit{if } \mathcal{S}_\beta = \emptyset \\
\quad\quad \textit{then } \mathtt{tt} \\
\quad\quad \textit{else} \\
\quad\quad\quad \textit{let } t = \min_{\sqsubseteq}(\mathcal{S}_\beta) \textit{ in} \\
\quad\quad\quad \textit{let } t' = \min_{\sqsubseteq}(\mathcal{S}_{\beta'}) \textit{ in} \\
\quad\quad\quad\quad t \sqsubseteq t' \wedge (t = t' \Rightarrow \mathcal{S}_\beta \backslash \{t\} \lesssim \mathcal{S}_{\beta'} \backslash \{t'\}), \\
\quad \lfloor \_ \rfloor = \lambda \beta : B.\ \lfloor \beta \rfloor, \\
\quad \lambda \beta : B.Gr(\beta) \\
\quad \rangle
\end{array}
$$

Fig. 4. Functor on named sets.

$$
\begin{array}{l}
T_2 \stackrel{\triangle}{=} \lambda H : \mathtt{NF}. \\
\quad \textit{let } S = T_1(\mathrm{dom}_H) \\
\quad \textit{and } D = T_1(\mathrm{cod}_H) \\
\quad \textit{and } h' = \lambda \beta \in S. \\
\quad\quad \langle \mathrm{cod}_H,\ \{\langle h_H(q), \ell, \pi, \sigma'; \sigma\rangle \mid \langle q, \ell, \pi, \sigma\rangle \in \mathcal{S}_\beta \ \wedge\ \sigma' \in \Sigma_H(q)\}\rangle \\
\quad \textit{and } h = \lambda \beta \in S.\mathrm{norm}\ h'(\beta) \\
\quad \textit{and } \Sigma = \lambda \beta \in S.\{\rho; \theta^{-1}_{h'(\beta)} \mid \rho \in Gr(h(\beta))\} \\
\quad \textit{in } \langle S, D, h, \Sigma\rangle
\end{array}
$$

Fig. 5. Functor on named functions.

Intuitively, Lemma 3.3 states that β "includes" the bundle resulting from its normalization. Sometimes we will 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 describes the functor T for the π-calculus. As usual [24], 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 in Fig. 4. When applied to a named set A, it returns a named set whose components are described below:

- the underlying set is characterized by the type of bundles over A;
- the order relation is the order on bundles induced by the order of A (Definition 3.6);
- the weight function is the function that gives the number of names appearing in the quadruples of the bundle;

Notice that T_1 requires the existence of a minimal quadruple; this is indeed the case because $\mathcal{S}_\beta$ is a finite set of quadruples, for any bundle β.

Fig. 5 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

- the domain is obtained by applying T_1 to dom_H;

- the codomain is obtained by applying T_1 to cod_H;
- the function $h_{T_2(H)}$ maps each bundle in the domain to a normalized bundle in the codomain;
- for each bundle β, the set of its name correspondences is obtained by composing symmetry $Gr(h_{T_2(H)}(\beta))$ with θ_β^{-1} to undo the normalization of names performed by computing *norm*.

We now prove that map T_2 is *functorial*, i.e., it preserves composition and identity.

Proposition 3.5. *Let* $H : \mathsf{NF}$, $K : \mathsf{NF}$ *be two named functions that can be composed. Then,*

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

Proof. First, we prove that both sides of Eq. (3) have the same domain and codomain.

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

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

Second, we show that $h_{T_2(H;K)} = h_{T_2(H);T_2(K)}$. By definition 3.4 and the definition of T_2, we have, for any $\beta \in \mathrm{dom}_{T_2(H)}$,

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

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

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

and, by Lemma 3.3,

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

Let us now consider function $h_{T_2(H;K)}$:

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

Hence, $h_{T_2(H;K)}(\beta) = \mathrm{norm}\ \langle \mathrm{cod}_{T_2(K)}, \mathcal{S} \rangle$ where, $\mathcal{S}$ contains all quadruples of the form $\langle h_K(h_H(q)), \ell, \pi, \bar{\sigma}; \hat{\sigma} \rangle$ such that

$$\langle q, \ell, \pi, \hat{\sigma} \rangle \in \mathcal{S}_\beta \quad \text{and} \quad \bar{\sigma} \in \Sigma_K(h_H(q)); \Sigma_H(q).$$

This, together with (4), implies that (for any β) $Q_\beta \subseteq \mathcal{S}_{h_{T_2(H;K)}(\beta)}$. Moreover, $h_{T_2(H);T_2(K)} \subseteq h_{T_2(H;K)}(\beta)$ follows by Lemma 3.2.

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

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

$$\Sigma_{T_2(H);T_2(K)}(\beta) = \Sigma_{T_2(K)}(h_{T_2(H)}(\beta)); \Sigma_{T_2(H)}(\beta),$$

hence

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

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

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

(equality (5) by associativity of composition). By Definition 3.2, we conclude from (5) that $\Sigma_{T_2(H);T_2(K)}(\beta) = \Sigma_{T_2H;K}(\beta)$. □

Proposition 3.5 shows that T_2 preserves composition. It remain to prove that identities are also preserved.

Proposition 3.6. *Map T_2 preserves identities.*

Proof. Given a named set A : NS, id_A : NF is the named functions such that $\mathrm{dom}_{\mathrm{id}\,A} = A$ and h_{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 $T_1(A)$. □

Proposition 3.7. *Map T_2 is functorial.*

3.2.4. From π-calculus to HD-automata

Following [18], we now construct the HD-automaton corresponding to the early semantics of the π-calculus. We assume existence of a function **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 $\leqslant$, where $x_i \leqslant x_j$ if, and only if, $i \leqslant j$.

In order to have a standard way to choose way the canonical transition (up to permutations of names) we borrow from [21] the definition of *representative transitions*.

Definition 3.7 (*Representative transition*). A 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 $y = \min(\mathcal{N} \backslash \mathrm{fn}(P))$;
- $\mu = xy$ and $y \in \mathrm{fn}(P) \cup \{\min(\mathcal{N} \backslash \mathrm{fn}(P))\}$.

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 only use representative transitions in HD-automata that correspond to π-agents. The following lemma ensures that these transitions are enough to capture agents' behaviour.

Lemma 3.4 (*Pistore [21, Lemma 7.6]*). *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 is obtained via 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:

$$Q_{D[P]} \stackrel{\triangle}{=} \bigcup_{P' \in S[P]} Q_{D[P']} \cup S[P],$$

where $S[P] \stackrel{\triangle}{=} \{\bar{P}\} \cup \{\overline{P'} \mid P \xrightarrow{\mu} P' \text{ is a representative transition } \wedge\ \mathbf{n}(P') = \langle \overline{P'}, \theta_{P'} \rangle\}$. Intuitively, $Q_{D[P]}$ is the set of (the canonical representative) agents that can be reached from P (through representative transitions). It is straightforward 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 the identity on $\mathrm{fn}(q)$ or is $\emptyset$, depending on $\mathrm{fn}(q) \neq \emptyset$.

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

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

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 $\Sigma_{K[P]}(q) = \{\rho; \theta_{\beta_q}^{-1} | \rho \in \mathcal{G}_{\mathrm{cod}_{K[P]}}(h_{K[P]}(q))\}$, 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:

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

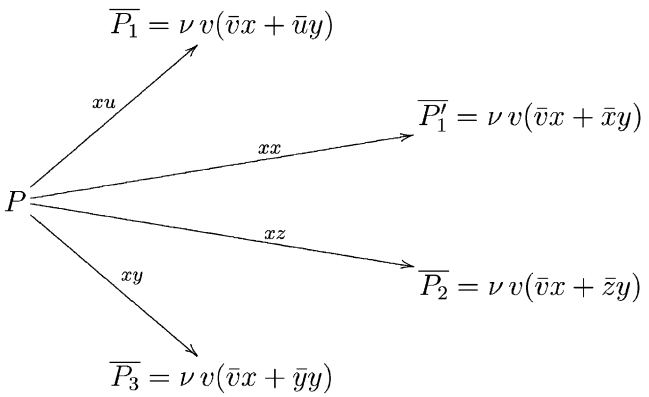
Agent P is *finitary* if $\max\{\deg(P') \mid P \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_i} P'\} < \infty$.

By taking advantage of the techniques introduced in [18,21] we have:

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

Proof. Easy. It is sufficient to mimic the proof of Theorem 47 of [18]. □

Example 3.3. We show how the functor acts by constructing the HD-automaton of agent $P \triangleq x(u).(\nu v)(\bar{v}z + \bar{u}y)$ 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 representative transitions:



Notice that P_1 and P_1' are the same up-to a bijective substitution; therefore they have the same canonical representation, namely $\overline{P_1} = \overline{P_2}$ and are represented by means of a unique state in the automaton. Transitions xx and xu are distinguished by a different σ in the automaton. Finally, the remaining representative transitions are $\overline{P_1} \xrightarrow{\bar{u}y} \mathbf{0}$, $\overline{P_2} \xrightarrow{\bar{z}y} \mathbf{0}$, and $\overline{P_3} \xrightarrow{\bar{y}y} \mathbf{0}$.

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 $\mathrm{fn}(P)$, and is used in the standard representative $\overline{P_1}$ of P_1.

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

This section specifies the minimization algorithm for HD-automata and proves that it converges on finite HD-automata. The minimization algorithm builds the minimal realization $\bar{H}$ of (finite) HD-automata by constructing (approximations of) the final coalgebra morphism. The kernel of $\bar{H}$ yields the equivalence classes where equivalent states are grouped together. 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.

Given a T-coalgebra $K : \mathtt{NF}$ with named set A as source, we let $\mathit{unit} : \mathtt{NS} = \langle \mathit{unit}, \lambda x, y : \mathit{unit}.\mathtt{tt}, \lambda x : \mathit{unit}.0, \emptyset \rangle$ to be the *empty* named set (i.e., the named set on the vacuum type *unit* having () as its unique element). The minimization algorithm is specified in a declarative way by the equations below.

$$\text{Initial approximation: } H_0 \stackrel{\triangle}{=} \langle A,\ \bot,\ \lambda q : A.(),\ \lambda q : A.\emptyset \rangle. \tag{6}$$

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

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

At each iteration, two cases can arise: (i) a class is splitted because the states that it contains are no longer considered equivalent or (ii) a new active name is discovered. The algorithm terminates when both these two cases do not occur. This is equivalent to saying that there H_{n+1} is *equal* to H_n, for some n.

Since we model iterations by means of named functions, we need to establish when two named functions "are the same".

Definition 3.8 (*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:

(1) $\ker h_{H_1} = \ker h_{H_2}$;
(2) $\forall q \in \mathrm{dom}_{H_1}.|h_{H_1}(q)|_{\mathrm{cod}_{H_1}} = |h_{H_2}(q)|_{\mathrm{cod}_{H_2}}$;
(3) $\forall q \in \mathrm{dom}_{H_1}.\mathcal{G}_{\mathrm{cod}_{H_1}}(h_{H_1}(q)) = \mathcal{G}_{\mathrm{cod}_{H_2}}(h_{H_2}(q))$.

The equivalence of named functions imposes constraints both 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. Notice that Definition 3.8 does not imply that H_1 and H_2 are the same named function when $H_1 \widetilde{=} H_2$ because the underlying functions h_{H_1} and h_{H_2} can differ each other.

Proposition 3.8. *$\widetilde{=}$ is an equivalence relation.*

In Definition 3.8, codomains of H_1 and H_2 do not play any role, indeed, only the symmetries of the images of elements q in the domain are required to be preserved. Intuitively, this means that we do not care of the "structure" of the elements in the codomains, but only whether H_1 and H_2 induce the same partition on their domains (condition 1) and names have the same meaning (conditions 2 and 3). Equivalence of named functions is the formal device exploited for expressing the halting condition of the iterative algorithm. Namely, the algorithm terminates when $H_{n+1} \widetilde{=} H_n$, for some n. This amounts to saying that, at the $(n+1)$th iteration, all the states remain in the same equivalence class they where in at the nth iteration (i.e., $\ker H_{n+1} = \ker H_n$, by Definition 3.8) and that the number of active names and symmetries remain unchanged (conditions 2 and 3 in Definition 3.8).

The proof of the convergence of the algorithm is based on the fact that T_2 is a monotone functor over a partial order set having finite chains only.

Definition 3.9 (*Relation $\preceq$*). 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 $Q_{\ker H_2}$;
- $\forall A \in Q_{\ker H_1}.\forall B \in Q_{\ker H_2}.A \cap B \neq \emptyset \Rightarrow |A|_{\ker H_1} \leqslant |B|_{\ker H_2}$;
- $\forall A \in Q_{\ker H_1}.\forall B \in Q_{\ker H_2}.\forall q \in A \cap B.\Sigma_{H_1}(q) \subseteq \Sigma_{H_2}(q)$.

Proposition 3.9. *$\preceq$ is a pre-order.*

Proof. Straightforward. □

Proposition 3.10. *Let H_1 and H_2 be two named functions, then*

$$H_1 \preceq H_2 \ \wedge \ H_2 \preceq H_1 \ \Rightarrow \ H_1 \widetilde{=} H_2. \tag{8}$$

Proof. The first two conditions of Definition 3.9 and the hypothesis of (8) implies the first two conditions of Definition 3.8. It remains to prove that the hypothesis implies the last condition of Definition 3.8.

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

$$\mathcal{G}_{\mathrm{cod}_{H_1}}(h_{H_1}(q)); \sigma \neq \mathcal{G}_{\mathrm{cod}_{H_2}}(h_{H_2}(q)); \sigma. \tag{9}$$

Notice that $H_1 \preceq H_2 \ \wedge \ H_2 \preceq H_1$ and Definition 3.9 imply $\Sigma_{H_1}(q) = \Sigma_{H_2}(q)$; and conditions on named functions imply that $\mathcal{G}_{\mathrm{cod}_{H_i}}(h_{H_i}(q))$; $\sigma = \Sigma_{h_{H_i}}(q)$ (for $i = 1, 2$), that contradicts (9). □

Proposition 3.11 (*Monotony of T*). *T_2 is monotone.*

Proof. Let $H_i : \mathsf{NF}$ ($i = 1, 2$) be such that $H_1 \preceq H_2$; we prove that $T_2(H_1) \preceq T_2(H_2)$. Let (for $i = 1, 2$) $S_i = \mathrm{dom}_{T_2(H_i)}$, $D_i = \mathrm{cod}_{T_2(H_i)}$, $h_i = h_{T_2(H_i)}$ 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 contradiction, 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, $|h_1(\beta)|_{D_1} \leqslant |h_2(\beta)|_{D_2}$ by construction; indeed, for any quadruple $t \in \mathcal{S}_\beta$, $|\tau(t)|_{\mathrm{cod}_{H_1}} \leqslant |\tau(t)|_{\mathrm{cod}_{H_2}}$ (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 ($i = 1, 2$)

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

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

  $$\mathrm{map}\ (\lambda\mu : \Sigma_H(\beta).\langle h_H(\tau_t), \ell_t, \pi_t, \sigma_t; \mu\rangle)\ \Sigma_H(\tau_t) \tag{10}$$

  to H_1 we obtaining a number of quadruples less or equal that the application of (10) to H_2. Therefore, $Gr(h_1(\beta)) \subseteq Gr(h_2(\beta))$ which imply that $\Sigma_1 \subseteq \Sigma_2$. □

Monotony is preserved by composition of named functions, as stated by the following proposition.

Proposition 3.12 (*Composition and ordering*). *Let H_1, H_2 be two named functions such that $H_1 \preceq H_2$. For any $K : \mathsf{NF}$ 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

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

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

$$\Sigma_{H_1}(h_K(q)); \Sigma_K(q) \subseteq \Sigma_{H_2}(h_K(q)); \Sigma_K(q),$$

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

This concludes the proof. □

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

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

Proof. First, observe that, by monotony of T and Proposition 3.12, 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. □

By definition, for any named function H, $T_2(H)$ and H differs each other because their codomains always differs. However, in Theorem 3.2 we implicitly refer to the equivalence on NF, i.e., $\widetilde{=}$ (Definition 3.8). Relation $\widetilde{=}$ is based on the notion of kernel of named functions, hence, the fix point is a named functions such that $\hat{H} \widetilde{=} F_K(\hat{H})$. 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) a bijective correspondence can be established between $\mathrm{dom}_{\hat{H}}$ and $\mathrm{dom}_{F_K(\hat{H})}$ such that order, weight and group components are preserved.

We can establish some basic properties of the outcome of the minimization algorithm that can be formally characterized.

We first prove the following theorem that the minimization algorithm does not "collapse" non bisimilar states.

Theorem 3.3. *Let $K[P]$ be the HD-automaton corresponding to a π-calculus finite control agent P. If $\hat{H}$ is the outcome of the minimization algorithm applied to $K[P]$ then any two states that are in the same equivalence class of $Q_{\ker \hat{H}}$ are bisimilar π-calculus agents*.

Proof. First, by construction, $\hat{H}$ is a named function such that $\mathrm{dom}_{\hat{H}} = Q_{D[P]}$ (the set of the processes reachable from P up to bijective renaming, as defined in 3.2.4). Indeed, $\ker \hat{H}$ induces a partition on $Q_{D[P]}$; two processes, say Q and R, are in the same class iff $h_{\hat{H}}(Q) = h_{\hat{H}}(R)$. If this is the case, then Q and R are early bisimilar. Indeed, if we let

$$\mathcal{R} = \{\langle Q, R\rangle | Q, R \in Q_{D[P]} \wedge h_{\hat{H}}(Q) = h_{\hat{H}}(R)\}, \tag{11}$$

then $\mathcal{R}$ is an early bisimulation. In order to prove (11), without loss of generality, we can consider only the representative transitions (see Lemma 3.4).

Let $t : Q \xrightarrow{\alpha} Q'$ be a representative transition. By construction, $K[P](Q)$ contains a transition representing t, say $\hat{t}$, built as described in Section 3.2.4. We distinguish two cases:

(1) If α is not an input transition, then $\hat{t} \in \mathcal{S}_{h_{\hat{H}}(Q)}$ because, at each iteration, the functor can only remove input transitions; hence, $\hat{t} \in \mathcal{S}_{h_{\hat{H}}(R)}$ and, therefore, $\hat{t}$ is in $K[R]$, that, by construction, means that $R \xrightarrow{\alpha} R'$, for some R'

(2) if α is an input transition, then $\mathcal{S}_{h_{\hat{H}}(Q)}$ either contains a transition corresponding to α or it contains a BIN transition that cover α. In both cases, we can apply the same reasoning above to construct $R \xrightarrow{\alpha} R'$.

We still have to prove the transfer property of bisimulation, namely, we must also guarantee $(Q', R') \in \mathcal{R}$. Let assume that such a transition does not exist; hence, $h_{\hat{H}}(Q') \neq h_{\hat{H}}(R')$ for all R' that are α-derivatives of R. By explicitly expanding the functor definition, at each step, we have

$$h_{H_{i+1}}(X) = \text{norm}\left\langle T_1(\text{cod}_{H_i}), \bigcup_{\langle X',\ell,\pi,\sigma\rangle \in \mathcal{S}_{h_K(X)}} \{\langle h_{H_i}(X'), \ell, \pi, \sigma'; \sigma\rangle \mid \sigma' : \Sigma_{H_i}(X')\}\right\rangle$$

then, $\mathcal{S}_{h_{\hat{H}}}(Q)$ contains a quadruple whose label and mapping correspond to α and whose target state is $h_{\hat{H}}(Q')$ and since $\mathcal{S}_{h_{\hat{H}}}(Q) = \mathcal{S}_{h_{\hat{H}}}(R)$ this quadruple must also be in $h_{\hat{H}}(R)$ meaning that for some α-derivative R' of R, $h_{\hat{H}}(Q') = h_{\hat{H}}(R')$, which gives the contradiction. □

Another property of the algorithm is that it does not distinguishes bisimilar processes.

Theorem 3.4. *Let P and Q be two bisimilar π-calculus agents and $\hat{H}$ the outcome of the minimization algorithm on $K[R]$, where $R = \tau.P + \tau.Q$. Then $h_{\hat{H}}(\bar{P}) = h_{\hat{H}}(\bar{Q})$, where $\mathbf{n}(P) = (\bar{P}, \theta_P)$ and $\mathbf{n}(Q) = (\bar{Q}, \theta_Q)$.*

Proof. First notice that $P \sim Q$ iff $R \sim \tau.P$ and that $\bar{P}$ and $\bar{Q}$ are both in $K[R]$ since they are τ-derivatives of R. We prove by induction that at each iteration H_i, $h_{H_i}(\bar{P}) = h_{H_i}(\bar{Q})$. The proof is trivial for the initial step. Assume that $h_{H_i}(\bar{P}) = h_{H_i}(\bar{Q})$ holds at the step $i > 0$. By expliciting the computation of the iteration step

$$h_{H_{i+1}}(\bar{P}) = \text{norm}\left\langle T_1(\text{cod}_{H_i}), \bigcup_{\langle \bar{P}',\ell,\pi,\sigma\rangle \in \mathcal{S}_{h_K(\bar{P})}} \{\langle h_{H_i}(\bar{P}'), \ell, \pi, \sigma'; \sigma\rangle \mid \sigma' : \Sigma_{H_i}(\bar{P}')\}\right\rangle,$$

$$h_{H_{i+1}}(\bar{Q}) = \text{norm}\left\langle T_1(\text{cod}_{H_i}), \bigcup_{\langle \bar{Q}',\ell,\pi,\sigma\rangle \in \mathcal{S}_{h_K(\bar{Q})}} \{\langle h_{H_i}(\bar{Q}'), \ell, \pi, \sigma'; \sigma\rangle \mid \sigma' : \Sigma_{H_i}(\bar{Q}')\}\right\rangle.$$

Assume that $t = \langle h_{H_i}(P'), \ell, \pi, \sigma\rangle \in \mathcal{S}_{H_{i+1}(\bar{P})} \backslash \mathcal{S}_{H_{i+1}(\bar{Q})}$ and that $P \xrightarrow{\alpha} P''$ (where $\mathbf{n}(P'') = (P', \theta_{P''})$) is the transition corresponding to the quadruple. We now reason by

case analysis:

- If no quadruple in $\mathcal{S}_{H_{i+1}(\bar{Q})}$ has label corresponding to α then $Q \stackrel{\alpha}{\not\rightarrow}$ and this contradicts the hypothesis $P \sim Q$.
- Let $h_{H_i}(P') \neq h_{H_i}(Q')$ hold for any Q' α-derivative of Q. Since $P \sim Q$, there is $Q \stackrel{\alpha}{\rightarrow} Q''$ such that $P'' \sim Q''$ and $\mathbf{n}(Q'') = (Q', \theta_{Q'})$ that, by induction, implies $h_{H_i}(P') = h_{H_i}(Q')$, yielding a contradiction.
- The last possibility is that the *norm* operation removes t from $\mathcal{S}_{h_{H_i(\bar{Q})}}$ while leaving it in $\mathcal{S}_{h_{H_{1+i}(\bar{P})}}$. This is not possible because it would mean that α is a free input xy and there is a bound input quadruple in $\mathcal{S}_{h_{H_{1+i}(\bar{Q})}}$ with a name correspondence function differs from σ just because $\theta_Q(x)$ is mapped on 0. However, in this case such a bound input quadruple would also belong in $\mathcal{S}_{h_{H_i(\bar{P})}}$ because P and Q are bisimilar and the target states of the transitions that bisimulates must have the same set of active names.

This concludes the proof. □

The proof of Theorem 3.4 builds the automaton for $\tau.P + \tau.Q$ instead of the simpler one for $P + Q$ because we need an automaton that has two states that "syntactically" correspond to P and Q.

Theorems 3.3 and 3.4 basically state that, for finite HD-automata, the minimization algorithm preserves the π-calculus early bisimulation and can be used for checking bisimilarity between processes.

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 minimization 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 minimization 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[14,15,28]; 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 parameterized using *functors*, that are functions from structures to structures: An ocaml

[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`.

functor constructs new modules by mapping modules of a given signature on structures of other signatures.

The following example (borrowed from [16]) 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 `S'` is the structure 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

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

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 $\lambda^{\rightarrow,\Pi,\Sigma}$; 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 [16] for a deeper discussion on this topic.

Fig. 5 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 normalization. The connections express typing relationships among the modules. For instance, since states in bundles and transitions must have the same type, then a connection exists between Bundle and Transitions modules.

The iterative construction of the minimal automaton is parameterized with respect to the modules of Fig. 5. 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

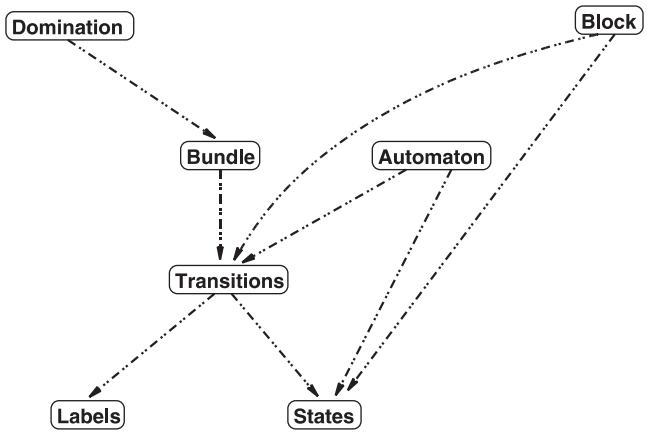


Fig. 6. Mihda Software Architecture.

software architecture. For instance, the architecture of Mihda has been exploited to provide minimization of both HD-automata and ordinary automata (up to strong bisimilarity).

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 $\lambda^{\rightarrow,\Pi,\Sigma}$ 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 l_i denotes its ith element (i.e., e_i). Finally, we write $\mathsf{e} \in \mathsf{l}$ 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 $\mathsf{e} \in \mathsf{x}$ 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 permutation 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*

$$\begin{pmatrix} 1 & \dots & n \\ i_1 & \dots & i_n \end{pmatrix}, \quad \rho = [i_1; ...; i_n].$$

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 ith 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 ith 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 lists of integers.

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

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

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 inhabitants of `quadtype` and, therefore, between automata and elements in `Automaton_t`.

Definition 4.2 (*Correspondence of quadruples*). Let qd be the quadruple $\langle q, \ell, \pi, \sigma\rangle$. 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*). The tuple (`q, qs, as, S`) *corresponds to* the named function $K = \langle Q, \mathcal{T}(Q), k : Q \to \mathcal{T}(Q), \Sigma\rangle$ iff, `qs` corresponds to Q; for each $t \in k(q)$ there exists `a` $\in$ `as` such that, if `a` = (`s`, (`lab`, `pi`, `sigma`), `t`), then `Qd((lab, pi, sigma), t)` corresponds to t, and, for each $\sigma \in \Sigma(q)$ there is `s` $\in$ `S` such that `s` corresponds to σ.

4.2. The main cycle

The generic step of the minimization algorithm ($H_{i+1} = K; T_2(H_i)$) can be explicitly written as $h_{H_{i+1}}(q) = \mathit{norm}\ \langle T_1(\mathrm{cod}_{H_i}), \beta\rangle$, where

$$\beta = \bigcup_{\langle q',\ell,\pi,\sigma\rangle \in \mathcal{S}_{h_K(q)}} \{\langle h_{H_i}(q'), \ell, \pi, \sigma'; \sigma\rangle \mid \sigma' : \Sigma_{H_i}(q')\}. \tag{12}$$

Following Eq. (12), we can compute $h_{H_{i+1}}(q)$ through the following steps:

(a) determine the bundle of q in the automaton, i.e., $h_K(q)$;
(b) for each quadruple $\langle q', \ell, \pi, \sigma\rangle$ in $h_K(q)$, 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) normalize the resulting bundle.

Mihda stores the representation of the minimized automaton at the ith 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)
```

The fields represent

- the name of the block (*id*); it is used to identify the block in order to construct the minimal automaton at the end of the algorithm,
- the states (*states*) considered equivalent with respect the equivalence relation used in the algorithm[6] (i.e., early bisimulation),
- the normalized bundle with respect to the block considered as state (*norm*),
- the list of names of the bundle in *norm* (*names*),
- the group of the block (*group*),
- the functions of the names of the bundle (Σ),
- the function (Θ^{-1}) that maps the names appearing in *norm* into the name of q.

Basically, $\Theta^{-1}(q)$ is the function which establishes a correspondence between the bundle of q and the bundle of the corresponding representative element in the equivalence class of the minimal automaton.

A graphical representation of a block is reported in Fig. 7. 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 Fig. 7 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 θ_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 Fig. 8. Step (a) is computed by the facility `Automaton.bundle` that filters all transitions of the automaton whose source corresponds to q. Fig. 8(a) shows that a state q is taken from a block and its bundle is computed.

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

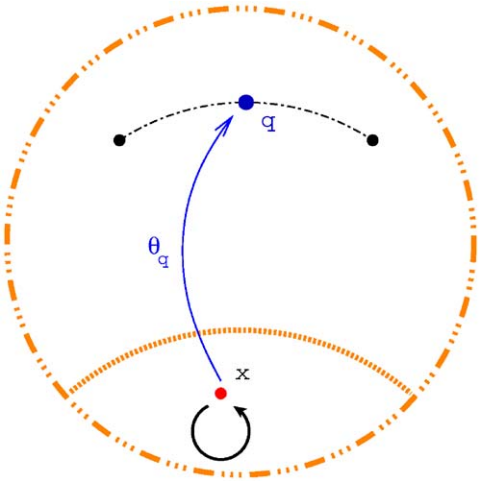


Fig. 7. Graphical representation of a block.

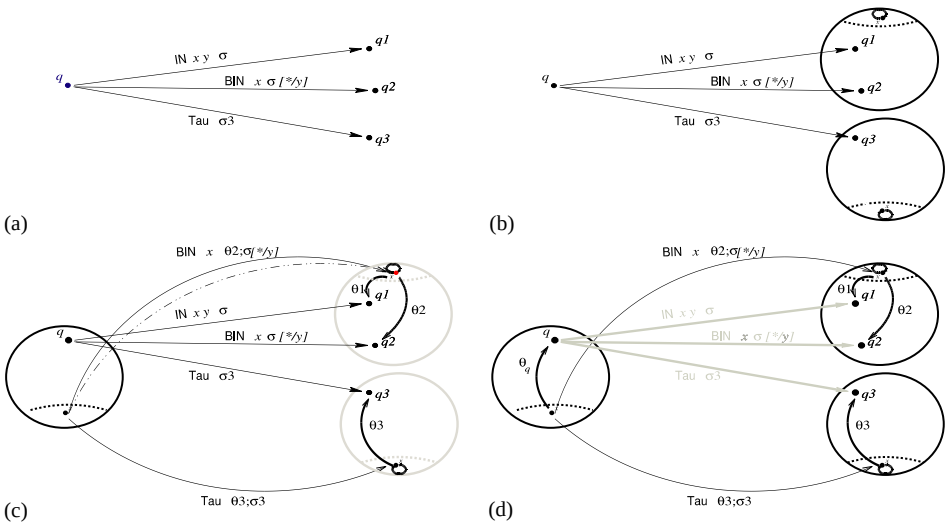


Fig. 8. Computing $h_{H_{i+1}}$: (a) step 1, (b) step 2, (c) step 3 and (d) step 4.

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 Fig. 8(b)).

Step (c) does not seem to correctly adhere to the corresponding step of Eq. (12). 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 Fig. 8(d) and is obtained via the function `normalize` 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 =
        let minimal =
          (Bundle.minimize red
            (Block.next (h_n blocks) (state_of blocks)
            (Automaton.bundle aut (List.hd (Block.states block))))) in
          Some (Block.split
            minimal
            (fun q →
              let normal =
                (Bundle.normalize
                  red
                  (Block.next (h_n blocks) (state_of blocks)
                    (Automaton.bundle aut q))) in
                Bisimulation.bisimilar minimal normal)
            block)
```

Let `block` be a block in the list `blocks`, function `split` computes `minimal` by minimizing 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 normalized 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 normalized 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 θ_q then `Some` θ_q is returned, otherwise `None` is returned.

We are now ready to comment on the main cycle of Mihda reported in Fig. 9. 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 the function `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`,

[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` [`f(e)`$_1$; ...; `e`$_h$] is the list [`f(e`$_1$`)`; ...; `f(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
```

Fig. 9. The main cycle of Mihda.

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 develops coalgebraic framework to specify HD-automata and their finite state verification techniques. The formal devices used in this work are coalgebras and $\lambda^{\rightarrow,\Pi,\Sigma}$, a polymorphic λ-calculus. On the one hand, coalgebras allow us to express transition systems in an elegant mathematical framework. On the other hand, $\lambda^{\rightarrow,\Pi,\Sigma}$ 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 minimization algorithm on finite HD-automata. Second, using $\lambda^{\rightarrow,\Pi,\Sigma}$ as a specification language and ocaml as the implementation language of Mihda, permits to point out the tight correspondence between the specification and the implementation.

From a programming perspective, our approach enjoys a high level of modularization. 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 minimizing both HD-automata and traditional automata; or else, automata can be minimized 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 preliminary results can be found in [6] while some experimental results of Mihda can be found in [5,27]; and seem quite promising. The π-calculus specification of the Handover Protocol (borrowed from [19,29]) has been minimized running Mihda on a machine equipped with an AMD Athlon™XP 1800+ dual processor with 1 G RAM. The time required for minimizing 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-minimized 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 relying on well-known results in coalgebras (e.g., [23,30]), different strategies for the convergence theorem (Theorem 3.2, p. 34) can be developed. However, the proofs given in this paper have two advantages: First, they are conceptually more simple and, second, they are based on those constructions used in the implementation thus providing hints for correctness of Mihda.

Acknowledgements

We would like to thank the anonymous referees for their comments and suggestions. We are very grateful to Marco Pistore for many interesting discussions on HD-automata and the minimization algorithm. We express our gratitude to Roberto Raggi who had a great role in the implementation of the algorithm. Finally, we would like to thank Marcello Bonsangue and Frank de Boer for the way they handled our submission.

References

[1] M. Abadi, A. Gordon, A calculus for cryptographic protocols: the spi calculus, Inform. Comput. 148 (1) (1999) 1–70.
[2] E. Clarke, J. Wing, Formal methods: state of the art and future directions, ACM Comput. Surveys 28 (4) (1996) 626–643.
[3] J. Fernandez, An implementation of an efficient algorithm for bisimulation equivalence, Sci. Comput. Programming 13 (2–3) (1990) 219–236.
[4] G. Ferrari, U. Montanari, M. Pistore, Minimizing transition systems for name passing calculi: a co-algebraic formulation, in: M. Nielsen, U. Engberg (Eds.), FOSSACS 2002, Lecture Notes in Computer Science, Vol. 2303, Springer, Berlin, 2002, pp. 129–143.
[5] G. Ferrari, U. Montanari, E. Tuosto, From co-algebraic specifications to implementation: the Mihda toolkit, in: F. de Boer, M. Bonsangue, S. Graf, W. de Roever (Eds.), Second International Symposium on Formal Methods for Components and Objects, Lecture Notes in Computer Science, Vol. 2852, Springer, Berlin, 2002, pp. 319–338.
[6] G. Ferrari, U. Montanari, E. Tuosto, B. Victor, K. Yemane, Modelling and minimising the fusion calculus using hd-automata, draft, January 2004.
[7] R. Focardi, R. Gorrieri, A classification of security properties, J. Comput. Security 3(1) (1995).
[8] M. Gabbay, A. Pitts, A new approach to abstract syntax involving binders, in: G. Longo (Ed.), Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99), IEEE Computer Society Press, Trento, Italy, 1999, pp. 214–224.

[9] P. Kanellakis, S. Smolka, CCS expressions, finite state processes and three problem of equivalence, Inform. Comput. 86 (1) (1990) 272–302.
[10] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[11] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, Cambridge, 1999.
[12] R. Milner, Theories for the global ubiquitous computer, in: Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science, Vol. 2987, Springer, Berlin, 2004, pp. 5–11.
[13] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Inform. Comput. 100(1) (1992) 1–40, 41–77.
[14] R. Milner, M. Tofte, Commentary on Standard ML, MIT Press, Cambridge, MA, 1991.
[15] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), The MIT Press, Cambridge, MA, 1997.
[16] J. Mitchell, Foundations for Programming Languages, MIT press, Cambridge, MA, 1996.
[17] U. Montanari, M. Pistore, History dependent automata, Technical Report, Computer Science Department, Università di Pisa, tR-11-98, 1998.
[18] U. Montanari, M. Pistore, π-calculus, structured coalgebras, and minimal HD-automata, in: M. Nielsen, B. Roman (Eds.), MFCS: Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 1983, Springer, Berlin, 2000, pp. 569–578; Theoret. Comput. Sci., to be published (extended version).
[19] F. Orava, J. Parrow, An algebraic verification of a mobile network, Formal Aspects Comput. 4 (5) (1992) 497–543.
[20] R. Paige, R. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (6) (1987) 973–989.
[21] M. Pistore, History dependent automata, Ph.D. Thesis, Computer Science Department, Università di Pisa, 1999.
[22] A. Pitts, M. Gabbay, A metalanguage for programming with bound names modulo renaming, in: R. Backhouse, J. Oliveira (Eds.), Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000, Lecture Notes in Computer Science, Vol. 1837, Springer, Berlin, 2000, pp. 230–255.
[23] J. Rutten, Universal coalgebra: a theory of systems, Theoret. Comput. Sci. 249 (1) (2000) 3–80.
[24] D. Rydeheard, R. Burstall, Computational Category Theory, Prentice-Hall, New York, 1988.
[25] D. Sangiorgi, A theory of bisimulation for the pi-calculus, Lecture Notes in Computer Science, Vol. 715 (2000).
[26] D. Sangiorgi, D. Walker, The π-Calculus: A Theory of Mobile Processes, Cambridge University Press, Cambridge, 2002.
[27] E. Tuosto, Non-functional aspects of wide area network programming, Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, May 2003, http://www.di.unipi.it/phd/tesi/tesi_2003/PhDthesis_Tuosto.ps.gz.
[28] J. Ullman, Elements of ML Programming, second ed., ML97 ed., Prentice-Hall, Englewood Cliffs, NJ, 1997.
[29] B. Victor, F. Moller, The mobility workbench—a tool for the π-calculus, in: D. Dill (Ed.), Proceedings of CAV'94, Lecture Notes in Computer Science, Vol. 818, Springer, Berlin, 1994, pp. 428–440.
[30] J. Worrell, Terminal sequences for accessible endofunctors, in: B. Jacobs, J. Rutten (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 19, Elsevier, Amsterdam, 1999.