

From Co-algebraic Specifications to Implementation: The Mihda Toolkit*

Gianluigi Ferrari, Ugo Montanari, Roberto Raggi, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa, Italy.
{giangi,ugo,raggi,etuosto}@di.unipi.it

Abstract. This paper describes the architecture of a toolkit, called *Mihda*, providing facilities to minimize labelled transition systems for name passing calculi. The structure of the toolkit is derived from the co-algebraic formulation of the partition-refinement minimization algorithm for HD-automata. HD-automata have been specifically designed to allocate and garbage collect names and they provide faithful finite state representations of the behaviours of π -calculus processes. The direct correspondence between the coalgebraic specification and the implementation structure facilitates the proof of correctness of the implementation. We evaluate the usefulness of *Mihda* in practice by performing finite state verification of π -calculus specifications.

1 Introduction

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

A fundamental property of automata is the possibility, given an automaton, to construct its canonical form: The minimal automaton. The theoretical foundations guarantee that the minimal automaton is indistinguishable from the original one with respect to many behavioural properties (e.g., bisimilarity of automata and behavioural 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 [8]. 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 state explosion: They yield a small state space, but still retain all the relevant information for the verification.

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

Global computing systems consists of networks of stationary and mobile components. The primary features of a global computing systems is 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 providing formal models and effective verification techniques to support verification of global computing systems, current software engineering technologies provide limited solutions to some of the issues discussed above. The problem of formal verification of global computing systems still requires considerable research and dissemination efforts.

History Dependent automata (HD-automata shortly) have been proposed in [14, 11, 4] as a new effective model for name passing calculi. Name passing calculi (e.g. the π -calculus [10, 9, 16]) are basically the best known and probably the most acknowledged models of mobility. Moreover, they provide a rich set of techniques for reasoning about mobile 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 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 distinguished mechanisms of name passing calculi.

HD-automata have been abstractly understood as automata over a permutation model, whose ingredients are a set of names and an action of its group of permutations (renaming substitutions) on an abstract set. This framework is sufficient to describe and reason about formalisms with name-binding operations. It has been incorporated into various kinds of transition systems that aim at providing syntax-free models of name-passing calculi [5, 6, 12, 15].

It is important to emphasize that names of states of HD-automata have *local meaning*. For instance, assume that $A(x, y, z)$ denotes an agent having three (free) names x , y and z . Then agent $A(y, x, z)$, obtained through the transformation which swaps names x and y , is *syntactically* different from $A(x, y, z)$. However, these two agents can be *semantically* represented by means of a single state q of a HD-automaton simply by considering a “swapping” operation on the local names corresponding to names x and y . More generally, states that differs only for renaming of their local names are identified in HD-automata. This property allows for a very compact representation of name passing calculi.

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

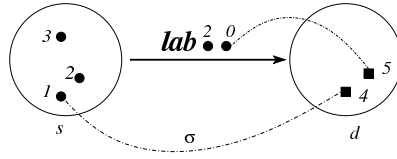


Fig. 1. A HD-automaton transition

4 and 5 which correspond to the old name 1 of s and to the fresh name 0, respectively. Notice that names 3 is discharged along such state transition.

HD-automata have a natural representation as coalgebras on a category of *named sets* and *named functions*. Elements of named sets are equipped with names which are defined up to groups of name permutations called *symmetries* [12]. General results concerning coalgebras guarantees the existence of the minimal HD-automata up to bisimilarity. In [4] two of the authors describe a declarative coalgebraic procedure to perform minimization of finite state HD-automata.

In this paper, we review the coalgebraic description of the minimization algorithm for HD-automata, and we illustrate its prototype implementation. This yields a toolkit, called *Mihda*, providing general facilities to minimize labelled transition systems for name passing calculi. The usefulness of the *Mihda* toolkit will be shown by performing finite state verification of π -calculus specifications.

The minimization algorithm has been specified by exploiting a type-theoretic notation rather than standard coalgebraic notations. The implementation data structures have been obtained by refining with efficiency considerations the type-theoretic formulation. The direct correspondence between the semantical structures and the implementation structures facilitates the design and the implementation of the toolkit. Moreover, it provides the formal machinery to perform the proof of correctness of the implementation.

Recently, several software engineering technologies have been introduced to support a programming paradigm where the web is exploited as a service distributor. By service we do not mean a monolithic web server but rather a component available over the web that others might use to develop other services. Conceptually, web services are stand-alone components that reside over the nodes of the network. Each web service has an interface which is network accessible through standard network protocols and describes the interaction capabilities of the service. The *Mihda* toolkit have been designed and made available as a WEB service. By a few clicks in a browser at the URL <http://jordie.di.unipi.it:8080/pweb/> the *Mihda* toolkit can be accessed remotely and its facilities can be evaluated directly over the WEB.

2 Preliminaries

This section sketches the main concepts on the coalgebraic representation of automata as a basis for finite state verification by semantic minimization. We

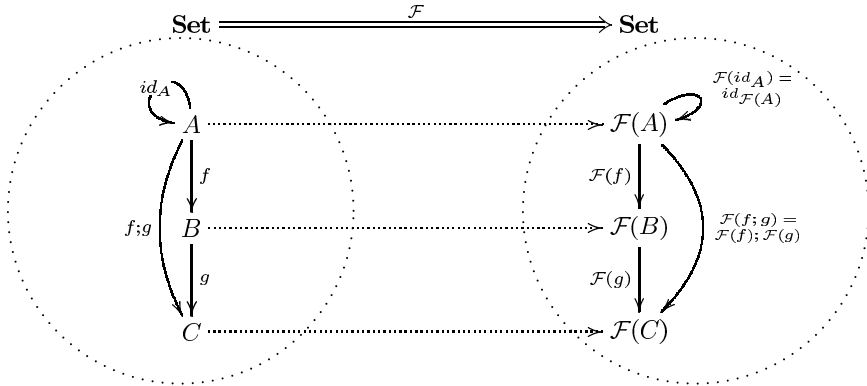


Fig. 2. Functor over **Set**

illustrate the approach by providing the coalgebraic specification of the minimization algorithm for ordinary labelled transition systems. Hereafter, we will use terms 'automaton' and 'labelled transition system' interchangeably.

An *automaton* A is a triple (S, L, \rightarrow) where S is the set of *states*, L is the set of *actions* or *labels* and $\rightarrow \subseteq S \times L \times S$ is the *transition relation*. Usually, one writes $s \xrightarrow{\ell} d$ to indicate $(s, \ell, d) \in \rightarrow$; s is the *source state* and d is the *destination* or *target state*.

Let id_A denote the identity function on set A and $f;g$ be the composition of functions f and g (when it is defined). An *endo-functor* \mathcal{F} (over category **Set**) is a mapping from sets to sets and from function to functions that preserves identity functions and function composition. Figure 2 gives a graphical representation of how a functor acts on sets and functions. If A is mapped on $\mathcal{F}(A)$ then id_A is associated to $id_{\mathcal{F}(A)}$ and, if f and g can be composed, then $\mathcal{F}(f)$ and $\mathcal{F}(g)$ can be composed as well. Moreover, the image through \mathcal{F} of the function composition $f;g$ is obtained by composing the images of functions f and g .

Definition 1 (\mathcal{F} -coalgebra). Let \mathcal{F} be an endo-functor on the category **Set**. A \mathcal{F} -coalgebra consists of a pair (A, α) such that $\alpha : A \rightarrow \mathcal{F}(A)$.

The duality between \mathcal{F} -coalgebras and \mathcal{F} -algebras (a function $\mathcal{F}(A) \rightarrow A$) consists in the fact that domain and codomain are “reversed”, namely, are arrows between the same objects but with opposite directions. Different directions can be interpreted as “construction” (induction) and “observation” (coinduction). The interested reader is referred to [7, 1].

Before specifying the coalgebraic description of the minimization algorithm we introduce some notations.

- Expression $Q : \mathbf{Set}$ denotes a set and $q : Q$ is synonym of $q \in Q$;
- **Fun** is the collection of functions among sets (the arrows of category **Set**). The function space over sets has the following structure:

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

- $h : A \xrightarrow{bij} B$ ($h : A \xrightarrow{inj} B$) explicitly states that function h is bijective (injective).

We shall use S_H, D_H and h_H to denote domain, codomain and mapping of an element of \mathbf{Fun} , respectively. A similar convention will be used throughout the paper to denote components of tuples.

Let $H, K \in \mathbf{Fun}$ be two functions, then the *composition* of H and K ($H;K$) is defined provided that $S_K = D_H$ and it is the function such that $S_{H;K} = S_H, D_{H;K} = D_K$, and $h_{H;K} = h_K \circ h_H$. Sometimes, we shall need to work with surjective functions. Hence we let \widehat{H} be the function given by $S_{\widehat{H}} = S_H, D_{\widehat{H}} = \{q' : D_H \mid \exists q : S_H, h_H(q) = q'\}$ and $h_{\widehat{H}} = h_H$.

Finite-state transition systems have been coalgebraically described by employing two ingredients: A set Q , that represents the state space, together with a function $K : Q \rightarrow \wp_{\text{fin}}(L \times Q)$ giving the transition relation; $K(q)$ is the set of pairs (ℓ, q') such that $q \xrightarrow{\ell} q'$.

In this paper, we shall work on a more concrete representation. In particular, we introduce a mathematical structure, called *bundle*, whose rôle is to provide a declarative specification of the concrete data structure storing all the transitions out of a given state. Indeed, each bundle details which states are reachable by performing certain actions.

Definition 2 (Bundles). *Let L be the set of labels, then a bundle β over L is a structure $\langle D : \mathbf{Set}, Step : \wp_{\text{fin}}(L \times D) \rangle$. Set D is the support of β .*

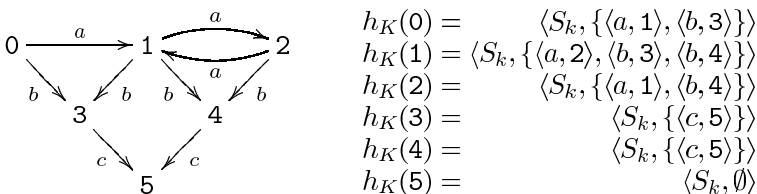
Given a fixed set of labels L, B^L denotes the collection of bundles and $\beta : B^L$ indicates that β is a bundle over L .

We now introduce the functor \mathcal{A} over the universe of sets and functions. The following clauses define \mathcal{A} :

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

Definition 3 (Transition systems as coalgebras). *Let L be a set of labels. Then a labelled transition system over L is a coalgebra for functor \mathcal{A} , namely it is a function K such that $D_K = \mathcal{A}(S_K)$.*

Example 1. A coalgebra K for functor \mathcal{A} represents a transition system where S_K is the set of states, and $h_K(q) = \beta$, with $D_\beta = S_K$. Let us consider a finite-state automaton and its coalgebraic formulation via the mapping h_K .



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

General results on coalgebras ensure the existence of the final coalgebra for a large class of functors. These results apply to our formulation of labelled transition systems. In particular, it is interesting to see the result of the iteration along the terminal sequence of functor \mathcal{A} .

Let K be a transition system, and let $H_0, H_1, \dots, H_{i+1}, \dots$ be the sequence of functions computed by

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

where H_0 is the unique function from S_K to the one-element set $\{*\}$ given by $S_{H_0} = S_K$; $D_{H_0} = \{*\}$; and $h_{H_0}(q : S_{H_0}) = *$.

Finiteness of \wp_{fin} ensures convergence of the iteration along the terminal sequence. We can say much more if the transition system is finite state. Indeed, if K is a finite-state transition system, then

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

Comparing the co-algebraic construction with the standard algorithm [8, 3] which constructs the minimal labelled transition system we can observe:

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

We now apply the iteration along the terminal sequence to the coalgebraic formulation of the transition system of Example 1. The initial approximation is the function H_0 defined as follows $H_0 = \langle S_{H_0} = S_K, D_{H_0} = \{*\}, h_{H_0}(q) = * \rangle$ and the first approximation H_1 is the map $h_{H_1} : q \mapsto \langle D_{H_0}, \{ \langle \ell, h_{H_0}(q') \rangle : q \xrightarrow{\ell} q' \} \rangle$ obtained by

$$T(H_0) \langle \{1, 2, 3, 4, 5\}, \{ \langle a, 2 \rangle, \langle b, 3 \rangle, \langle b, 4 \rangle \} \rangle = \langle \{*\}, \{ \langle a, * \rangle, \langle b, * \rangle \} \rangle$$

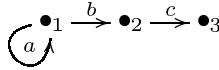
We obtain the function h_{H_1} and the destination state $D_{H_1} = \{\beta_1, \beta_2, \beta_3\}$ as detailed below.

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

A further iteration yields:

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

Since $D_{H_2} \equiv D_{H_1}$ the iterative construction converges, thus providing the minimal labelled transition system depicted as



where $\bullet_1 = \{0, 1, 2\}$, $\bullet_2 = \{3, 4\}$ and $\bullet_3 = \{5\}$.

3 HD-Automata for π -Agents

This section outlines the representation of HD-automata as coalgebras over the concrete permutation algebra of named sets.

Let \mathcal{N} be an infinite countable set of names ranged over by v and let \mathcal{N}^* be the set $\mathcal{N} \cup *$ where $*$ $\notin \mathcal{N}$ is a distinguished name. The distinguished name $*$ will be used to model name creation. We also assume a total order $<$ on \mathcal{N}^* (for instance, $<$ can be the lexicographic order on \mathcal{N} and $\forall v \in \mathcal{N} : * < v$).

Table 1 displays the definitions of *named sets*, *named functions*, and composition of named functions. In Table 1, the general product \prod is employed (as usual in type theory) to type functions f such that the type of $f(q)$ is dependent on q . Intuitively, a named set represents a set of states equipped with a mechanism to give local meaning to names occurring in each state. In particular, function $|_|_$ yields the number of local names of states. Moreover, the permutation group $G_A(q)$ allows one to describe directly the renamings that do not affect the behaviour of q , i.e., symmetries on the local names of q . For technical reasons, we assume that states are totally ordered.

By convention we write $\{q : Q_A\}_A$ to indicate the set $\{v_1, \dots, v_{|q|_A}\}$ and we use \mathbf{NSet} to denote the universe of named sets.

As in the case of standard transition systems, named functions are used to determine the possible transitions of a given state. Intuitively, $h_H(q)$ yields the

Table 1. Named sets, Named Functions and Composition of Named Functions

<p>Named set A <i>named set</i> A is a structure</p> $A = \langle Q : \mathbf{Set}, _{-} : Q \longrightarrow \omega, \leq : \wp(Q \times Q), G : \prod_{q \in Q} \wp(\{v_1..v_{ q }\} \xrightarrow{bij} \{v_1..v_{ q }\}) \rangle$ <p>where $\forall q : Q_A, G_A(q)$ is a permutation group and \leq_A is a total ordering.</p>
<p>Named function A <i>named function</i> H is a structure</p> $H = \langle S : \mathbf{NSet}, D : \mathbf{NSet}, h : Q_S \longrightarrow Q_D, \Sigma : Q_S \longrightarrow \wp(\{h(q)\}_D \xrightarrow{inj} \{q\}_S) \rangle$ <p>where $\forall q : Q_{S_H}, \forall \sigma : \Sigma_H(q)$,</p> <ol style="list-style-type: none"> 1. $G_{D_H}(h_H(q)); \sigma = \Sigma_H(q)$ and 2. $\sigma; G_{S_H}(q) \subseteq \Sigma_H(q)$.
<p>Composition of named functions Named functions can be composed in the obvious way. Let H and K be named functions. Then $H;K$ is defined only if $D_H = S_K$, and</p> $S_{H;K} = S_H, \quad D_{H;K} = D_K, \quad h_{H;K} : Q_{S_H} \longrightarrow Q_{D_K} = h_H;h_K,$ $\Sigma_{H;K}(q : Q_{S_H}) = \Sigma_K(h_H(q)); \Sigma_H(q)$ <p>Let H be a named function, \widehat{H} denotes the surjective component of H:</p> <ul style="list-style-type: none"> - $S_{\widehat{H}} = S_H$ and $Q_{D_{\widehat{H}}} = \{q' : Q_{D_H} \mid \exists q : Q_{S_H}. h_H(q) = q'\}$, - $q _{D_{\widehat{H}}} = q _{D_H}, G_{D_{\widehat{H}}}(q) = G_{D_H}(q), h_{\widehat{H}}(q) = h_H(q)$ and $\Sigma_{\widehat{H}}(q) = \Sigma_H(q)$

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

Table 2. Bundles: the π -calculus case

<p>Bundles A bundle β consists of the structure</p> $\beta = \langle \mathcal{D} : \mathbf{NSet}, Step : \wp(qd \mathcal{D}) \rangle$ <p>where $qd \mathcal{D}$ is the set of <i>quadruples</i> of the form $\langle \ell, \pi, \sigma, q \rangle$ given by</p> $qd \mathcal{D} = \{ \langle \ell : L_\pi, \pi : \mathcal{N} \xrightarrow{inj} \{v_1..\}, \sigma : \prod_{\ell \in L_\pi} \{q\}_{\mathcal{D}} \xrightarrow{inj} Q\ell, q : Q_{\mathcal{D}} \rangle \}.$ <p>and</p> $Q\ell = \begin{cases} \mathcal{N}^* & \text{if } \ell \in \{BOUT, BIN\} \\ \mathcal{N} & \text{if } \ell \notin \{BOUT, BIN\} \end{cases}$ <p>under the constraint that $G_{\mathcal{D}_\beta}(q); S_q = S_q$, where $S_q = \{ \langle \ell, \pi, \sigma, q \rangle \in Step_\beta \}$ and $\rho; \langle \ell, \pi, \sigma, q \rangle = \langle \ell, \pi, \rho; \sigma, q \rangle$.</p>
<p>Bundle names Let β be a bundle. Function $\llbracket \beta \rrbracket : B \rightarrow \mathcal{N}$, mapping each bundle to the set of its names, is defined by</p> $\llbracket \beta \rrbracket = \bigcup_{\langle \ell, \pi, \sigma, q \rangle \in Step_\beta} rng(\pi) \cup rng(\sigma) \setminus \{*\}$ <p>where rng yields the range of functions. We only consider bundles β such that $\llbracket \beta \rrbracket$ is finite and we let β to indicate the number of names which occur in the bundle β (i.e. $\beta = \llbracket \beta \rrbracket$).</p>

3.1 Bundles over π -Calculus Actions

To represent the minimization algorithm for the early semantics of π -calculus [10], the notion of bundle must be enriched. Labels of transitions must distinguish among the different meanings of names occurring in π -calculus actions, namely synchronization, bound/free output and bound/free input. The set of π -calculus labels L_π is $\{TAU, BOUT, OUT, BIN, IN\}$. We specify two different labels for input actions: Label BIN is used when the input transition exposes a fresh name, while label IN handles the case of an input transition that exposes a name of the source state of the transition. Labels in L_π have *weights*. The weight map $|-| : L_\pi \rightarrow \{\emptyset, \{1\}, \{1, 2\}\}$ is defined as:

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

and associates the set of indexes of distinct names the label refers to.

A bundle on π -labels is defined as in Table 2. Table 2 illustrates definitions of *bundles* and *names of bundles*. As it is the case for ordinary automata, the *Step* component of a bundle specifies the data structure that contains the set of successor states for a given source state. More precisely, if $\langle \ell, \pi, \sigma, q \rangle \in qd \mathcal{D}$,

then q is the destination state; ℓ is the label of the transition; π associates to the label the names observed in the transition; and σ states how names in the destination state are related to the names in the source state. According to the definition of σ in Table 2, a name in a destination state of a quadruple is mapped on the distinguished name $*$ only on transitions where a new name is created (i.e. transitions labelled by *BOUT* or *BIN*).

In order to exploit named functions for representing HD-automata it is necessary to equip the set of bundles B with a named set structure. In other words we must define

- a total order on bundles,
- a function that maps a bundle to its number of names,
- a group of permutations over those names.

The names of a bundle are the names (different from $*$) that appear either in the labels or in the range of σ 's of the quadruples of the bundle. Without loss of generality, we can assume that a total order on states and labels exist. Hence, quadruples are totally ordered¹. The order over quadruples yields an ordering \sqsubseteq over bundles.

The group of $\beta : B^{L\pi}$ is the set of permutations $\theta : \{\beta\} \xrightarrow{bij} \{\beta\}$ such that $\beta; \theta = \beta$ where $\beta; \theta$ is defined as $\langle \mathcal{D}_\beta, \{\langle \ell, \pi; \theta, \sigma; \theta, q \rangle \mid \langle \ell, \pi, \sigma, q \rangle : \beta \} \rangle$.

3.2 Normalizing Bundles

In the minimization algorithm two states belong to the same block (partition) whenever they have the “same” bundles. Hence, the most crucial construction on bundles is the *normalization* operation. This operation is necessary for two different reasons. The first reason is that there are different equivalent ways for picking up the step components (i.e. quadruples $\langle \ell, \pi, \sigma, q \rangle$) of a bundle. The second (more important) reason consists of removing from the step component of a bundle all the *redundant* input transitions. Indeed, redundant transitions occur when a HD-automaton is built from a π -calculus agent. During this phase, it is not possible to decide which free input transitions are required, and which transitions are irredundant². The solution to this problem consists of adding a superset of the required free input transitions and to exploit a reduction function to remove the unnecessary ones during the minimization phase. Consider for instance the case of a state q having only one name v_1 and assume that the following two tuples appear in a bundle:

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

Then, the *IN* transition is redundant if y is not active in q as it expresses exactly the same behaviour of the second tuple, except that a “free” input transition

¹ For instance, we can assume the lexicographic order of labels, states and names.

² In the general case, to decide whether a free input transition is required it is as difficult as to decide the bisimilarity of two π -calculus agents.

is used rather than a “bound” one. Hence, the transformation removes the first tuple from the bundle. During the iterative execution of the minimization algorithm, bundles are split; hence the set of redundant transitions of bundles decreases. Hence, when the iterative construction terminates, only those free inputs that are really redundant have been removed from the bundles.

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

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

where $\sigma; \{y \rightarrow *\}$ is the function equal to σ on any name different from y and that assigns $*$ to y . Second, the *normalization* function $norm(\beta)$ is defined as follows:

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

where $an_\beta = \{\!\!| red(\beta) \!\!\}$ is the active names of β and \min_{\sqsubseteq} is the function that, when applied to $Step_\beta$, returns the step of the minimal bundle (with respect to order \sqsubseteq) among those obtained by permuting names of β in all possible ways. More precisely, given a bundle $\bar{\beta}$, $\min_{\sqsubseteq} \bar{\beta}$ is the minimal bundle in the set $\{\bar{\beta}; \theta \mid \theta : \{\!\!| \bar{\beta} \!\!\} \xrightarrow{bij} \{\!\!| \bar{\beta} \!\!\}\}$, with respect to the total ordering \sqsubseteq of bundles over \mathcal{D} . The order relation \sqsubseteq is used to define the canonical representatives of bundles and relies on the order of quadruples. Hereafter, we use $perm(\beta)$ to denote the canonical permutation that associates $Step_{norm(\beta)}$ and $Step_\beta \setminus \{\langle IN, xy, \sigma, q \rangle \mid y \notin an_\beta\}$.

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

Finally, we need a construction which extracts in a canonical way a group of permutations out of a bundle. Let β be a bundle, define $Gr \beta$ to be the set $\{\rho \mid Step_\beta; (\rho[*/*]) = Step_\beta\}$. It can be proved that $Gr \beta$ is a group of permutations.

3.3 The Minimization Algorithm

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

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

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

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

Notice that functor \mathcal{T} maps every named set A into the named set $\mathcal{T}(A)$ of its *normalized* bundles. A named function H is mapped into a named function $\mathcal{T}(H)$ in such a way that every corresponding pair $(q, h_H(q))$ in h_H is mapped into a set of corresponding pairs $(\beta norm(\beta'))$ of bundles in $h_{\mathcal{T}(H)}$. The quadruples of bundle β' are obtained from those of β by replacing q with $h_H(q)$ and by saturating with respect to the set of name mappings in $\Sigma_H(q)$. The name mappings in $\Sigma_{\mathcal{T}(H)}(\beta)$ are obtained by transforming the permutation group of bundle $norm(\beta')$ with the inverse of the canonical permutation of β' and with a fixed injective function inj mapping the set of names of $norm(\beta')$ into the set of names of β , defined as $i < j, inj(v_i) = v_{i'}$ and $inj(v_j) = v_{j'}$ implies $i' < j'$. Without bundle normalization, the choice of β' among those in $\beta'; \theta$ would have been arbitrary and not canonical with the consequence of mapping together fewer bundles than needed.

Definition 4 (Transition systems for π -agents). *A transition system over named sets and π -actions is a named function K such that $D_K = \mathcal{T}(S_K).$*

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

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

We will often use the notation $q \xrightarrow{\ell, \pi, \sigma} q'$ to denote the “representative” transitions from agent q that are used in the construction of the HD-automaton.

We can now define the function K .

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

The minimal HD-automata is built by an iterative procedure on K : the iteration along the terminal sequence. The formula which details the iterative construction is given by

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

If K is a finite state HD-automaton. Then

The initial approximation, H_0 , is defined as follows:

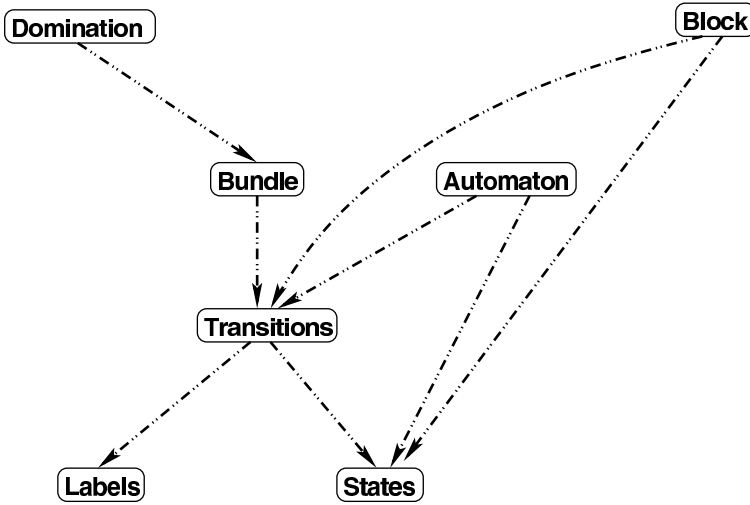


Fig. 3. Mihda Software Architecture

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

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

4 The Mihda Toolkit

The previous sections outlined the coalgebraic foundation for the finite state verification of name passing process calculi. It remains to show that this theory can be effectively used as a basis for the design and development of effective and usable verification toolkits. This section and the one following explore this issue by describing our experience in designing, implementing and experimenting a minimization toolkit, called Mihda, for verifying finite state mobile systems represented in the π -calculus.

The Mihda toolkit cleanly separates facilities that are language-specific (parsing, transition system calculation) from those that are independent from the calculus notation (bisimulation) in order to ease modifications. The toolkit has been implemented in `ocaml`. Indeed, the partition refinement algorithm has been specified in a “type-theoretic” style and the underlying type system makes use of parametric polymorphism. The type system of `ocaml` offers all the necessary features for handling these kind of types. Figure 3 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 modules **Bundle** and **Transitions**.

Notice that the iterative construction of the minimal automaton is parameterized with respect to the modules of Figure 3. Indeed, the same algorithm can be applied to different kind of automata and bisimulation, provided that these automata match the constraints on types imposed by the software architecture. For instance, the architecture of **Mihda** has been exploited to provide minimization of both HD-automata and ordinary automata (up to strong bisimilarity).

4.1 The Main Cycle

We have already pointed out that the iterative step of the minimization algorithm can be represented in a functional style form as follows:

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

We compute $h_{H_{i+1}}(q)$ through the following steps:

- (a) determine the bundle of state q ;
- (b) for each quadruple $\langle \ell, \pi, \sigma, q' \rangle$ in this bundle, apply h_{H_i} to q' , the target state of the quadruple (yielding the bundle of q' in the previous iteration of the algorithm);
- (c) left-compose symmetry $\sigma' \in \Sigma(q')$ with σ ;
- (d) normalize the resulting bundle.

In the **Mihda** implementation the value of the i -th iteration (i.e. h_{H_i}) is stored in a list of *blocks* which are the crucial data structures of **Mihda**. Blocks implements the action of the functor on states of the automata and contain all those information needed to compute the iteration steps of the algorithm expressed in a set theoretic framework. Blocks represent both (finite) named functions and partitions of an automaton (at each iteration of the algorithm). When the algorithm terminates, each block will correspond 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 *
     $\Sigma$     : (State.t  $\rightarrow$  (int * int) list list) *
     $\Theta^{-1}$  : (State.t  $\rightarrow$  (int * int) list)

```

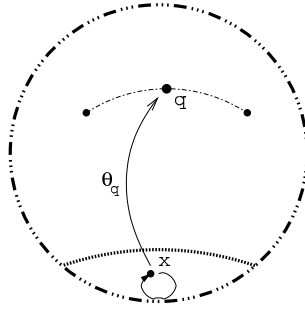


Fig. 4. Graphical representation of a block

Field *id* is the name of the block and is used to identify the block in order to construct the minimal automaton at the end of the algorithm. Field *states* contains the states which are considered equivalent. The remaining fields represent

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

We pictorially represent (some components of) a block as in Figure 4: The upper elements are the states in the block, while the element *x* is the “representative state”, namely it is a graphical representation of the block as a state. For each state *q* a function θ_q maps names of *x* into the names of *q*. Function θ_q describes “how” the block approximates the state *q* at a given iteration. The circled arrow on *x* aims at recording that a block also has symmetries on its names. Bundle *norm* of block *x* is computed by exploiting the ordering relations over names, labels and states. A graphical representation of steps (a)-(d) above in terms of blocks is illustrated in Figure 5.

Step (a) is computed by the facility `Automaton.bundle` that filters all transitions of the automaton whose source corresponds to *q*. Figure 5(a) shows that a state *q* is taken from a block and its bundle is computed.

Step (b) is obtained by applying facility `Block.next` to the bundle of *q*. The operation `Block.next` substitutes all target states of the quadruples with the corresponding current block and computes the new mappings (see Figure 5(b)).

Step (c) does not seem to correctly adhere to the corresponding step of equation 1. However, if we consider that θ functions are computed at each step by composing symmetries σ ’s we can easily see that θ functions exactly play the rôle of σ ’s.

Finally, step (d) is represented in Figure 5(d) and is obtained via the function `Bundle.normalize`.

The main step of the minimization algorithm is the function `split` that computes, at each iteration, the current partition (the list of blocks).

```

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

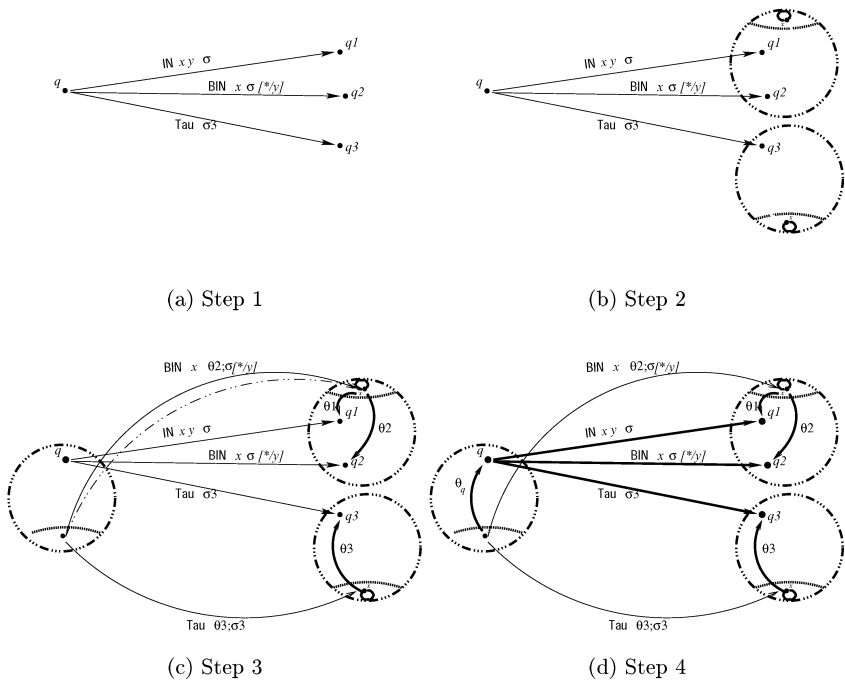


Fig. 5. Computing $h_{H_{i+1}}$

```

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. 6. The main cycle of Mihda

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 `Bisimulation.bisimilar`. 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 Figure 6. Let `k = (start, states, arrows)` be a HD-automaton. When the algorithm starts, `blocks` is the list that contains a single block collecting all the states of the automata `k`.

At each iteration, the list of blocks is splitted, as much as, possible by `split_iter` that returns a list of `buckets` which have the same fields of a block apart from the name, symmetries and the functions mapping names of destination states into names of source states. Essentially, the split operation checks if two states in a block are equivalent or not. States which are no longer equivalent to the representative element of the block are removed and inserted into a bucket. Then, by means of `Block.close_block`, all buckets are turned into blocks which are assigned to `blocks`. Finally, the termination condition `stop` is evaluated. This condition is equivalent to say that a bijection can be established between `oldblocks` (that corresponds to D_i) and `blocks` (corresponding to D_{i+1}). This

condition reduces to test whether `blocks` and `oldblocks` have the same length and that blocks at corresponding positions are equal.

5 Verifying Mobile Systems with Mihda

In this section we discuss some experimental results of `Mihda` in the analysis of mobile systems. In particular, we consider the π -calculus specification of the Handover Protocol for Mobile Telephones borrowed from that given in [17] (which has been in turn derived from that in [13]).

The π -calculus specification of the `GSM` is

```
define GSM(in,out) =
  (tca)(ta)(ga)(sa)(aa)(tcp)(tp)(gp)(sp)(ap)
  | ( Car(ta,sa,out),
    Base(tca,ta,ga,sa,aa),
    IdleBase(tcp,tp,gp,sp,ap),
    Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap))
```

`Centre` receives messages from the environment on channel `in`; these input actions are the only observable actions performed by `Centre`. Module `Car` sends the messages to the end user along the channel `out`; these outputs are the only visible actions performed by the `Car`. Modules `Centre` and `Car` interact via the base corresponding to the cell in which the car is located. The specification of modules `Car`, `Base`, `IdleBase` and `Centre` is reported in Table 3. The behaviour of the four modules is briefly summarized below:

- `Car` brings a `MobileStation` and travels across two different geographical areas that provide services to end users;
- `Base` and `IdleBase` are Base Station modules; they interconnect the `MobileStation` and the `MobileSwitching Centre`.
- `Centre` is a `MobileSwitching centre` which controls radio communications within the whole area composed by the two cells;

The protocol starts when `Car` moves from one cell to the other. Indeed, `Centre` communicates to the `MobileStation` the name of the base corresponding to the new cell. The communication of the new channel name to the `MobileStation` is performed via the current base. All the communications between the `MobileSwitching centre` and the `MobileStation` are suspended until the `MobileStation` receives the names of the new transmission channels. Then the base corresponding to the new cell is activated, and the communications between the `MobileSwitching centre` and the `MobileStation` continue through the new base.

In Table 4 we report the results of `Mihda` on two different versions of the protocols. The first row of the table corresponds to the version discussed above. The second line gives the figures on a version of the `GSM` protocol that models the `MobileSwitching` and `MobileStation` modules in a more realistic way. Indeed, the 'full' version exploits a protocol for establishing whether or not the car is crossing the boundary of a cell and entering the other cell.

Table 3. π -calculus specification of GSM modules

```

define Car(talk,switch,out) =
  talk?(msg).out!msg.Car(talk,switch,out) +
  switch?(t).switch?(s).Car(t,s,out)

define Base(talkcentre,talkcar,give,switch,alert) =
  talkcentre?(msg).talkcar!msg.
  Base(talkcentre,talkcar,give,switch,alert)
+
  give?(t).give?(s).switch!t.switch!s.give!give.
  IdleBase(talkcentre,talkcar,give,switch,alert)

define IdleBase(talkcentre,talkcar,give,switch,alert) =
  alert?(empty).Base(talkcentre,talkcar,give,switch,alert)

define Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap) =
  in?(msg).tca!msg.Centre(in,tca,ta,ga,sa,aa,tcp,tp,gp,sp,ap)
+
  tau.ga!tp.ga!sp.ga?(empty).ap!ap.
  Centre(in,tcp,tp,gp,sp,ap,tca,ta,ga,sa,aa)

```

Table 4. Mihda at work

Protocol	Time to compile	States	Transitions	Time to minimize	States	Transitions
GSM small	0m 0.931s	211	398	0m 4.193s	105	197
GSM full	0m 8.186s	964	1778	0m 54.690s	137	253

The results are obtained by running Mihda on an AMD AthlonTMXP 1800+ dual processor with 1Giga RAM. The time for minimizing the automata is very contained. The results on the GSM seem very promising. Indeed, the size of the minimal automata in terms of states and transitions is smaller than their non-minimized version. In the case of GSM small the size of the minimal automaton is the half or the automaton obtained by compiling the original specification; while, in version GSM full, states and transitions are reduced of a factor 8.

6 Conclusion

This paper has provided an overview of a foundational model for the finite state verification of global computing systems and has showed how efficient tool supports can be derived from it. We are currently extending 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). To improve efficiency, we plan to incorporate software supports for symbolic approaches based on Binary Decision Diagrams.

References

1. Peter Aczel. Algebras and coalgebras. In Roy Backhouse, Roland Crole and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 3, pages 79–88. Springer Verlag, April 2002. Revised Lectures of the Int. Summer School and Workshop.
2. Edmund M. Clarke and Jeanette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
3. Jean Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1990.
4. GianLuigi Ferrari, Ugo Montanari, and Marco Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In Mogens Nielsen and Uffe Engberg, editors, *FOSSACS 2002*, volume LNCS 2303, pages 129–143. Springer Verlag, 2002.
5. Marcelo Fiore, Gordon G. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1999.
6. Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1999.
7. Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1996.
8. Paris C. Kanellakis and Scott A. Smolka. Ccs expressions, finite state processes and three problem of equivalence. *Information and Computation*, 86(1):272–302, 1990.
9. Robin Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
10. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
11. Ugo Montanari and Marco Pistore. History dependent automata. Technical report, Computer Science Department, Università di Pisa, 1998. TR-11-98.
12. Ugo Montanari and Marco Pistore. π -calculus, structured coalgebras and minimal hd-automata. In *Mathematical Foundations of Computer Science 2000*, volume 1893. Springer, 2000.
13. Fredrik Orava and Joachim Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4(5):497–543, 1992.
14. Marco Pistore. *History dependent automata*. PhD thesis, Computer Science Department, Università di Pisa, 1999.
15. Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction, 5th International Conference, MPC2000*, volume 1837. Springer, 2000.
16. Davide Sangiorgi and David Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.
17. Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π -calculus. In David Dill, editor, *Proceedings of CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.