Realistic Implementation of Message Sequence Charts

Claude Jard, Rouwaida Abdallah, Loïc Hélouët

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Realistic Implementation of Message Sequence Charts

Claude Jard — Rouwaida Abdallah — Loïc Hélouët

N° 7597

March 2011

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7597--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Realistic Implementation of Message Sequence Charts

Claude Jard , Rouwaida Abdallah , Loïc Hélouët

Thème COM — Systèmes communicants
Équipes-Projets DistribCom

Rapport de recherche n° 7597 — March 2011 — 23 pages

Résumé : Ce travail étudie le problème de la synthèse de programmes à partir des spécifications décrites par des High-level Message Sequence Charts. Nous montrons d'abord que, dans le cas général, la synthèse par une simple projection sur chaque composante du système permet plus de comportements dans l'implémentation que dans la specification. Nous montrons ensuite que les comportements supplémentaires viennent d'une perte d'ordre entre les messages au moment de la projection, et que ces nouveaux comportements peuvent être évités en ajoutant des contrôleurs de communication qui interceptent les messages et qui y ajoutent des informations de contrôle avant de les envoyer au processus, préservant ainsi l'ordre décrit dans la spécification initiale.

Mots-clés : Message Sequence Charts, Communicating Finite State Machines, génération de code, Promela, Sofat

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Realistic Implementation of Message Sequence Charts

Abstract: This work revisits the problem of program synthesis from specifications described by High-level Message Sequence Charts. We first show that in the general case, synthesis by a simple projection on each component of the system allows more behaviors in the implementation than in the specification. We then show that differences arise from loss of ordering among messages, and show that behaviors can be preserved by addition of communication controllers, that intercept messages to add stamping information before resending them, and deliver messages to processes in the order described by the specification.

Key-words: High-level Message Sequence Charts, Communicating Finite State Machines , code generation, Promela, Sofat

Table des matières

1 Introduction

Message Sequence Charts (MSCs for short)) [13] is a formal language based on composition of finite communication sequences. MSCs are made of two layers of specification. At the lowest level, basic MSCs describe the behavior of a finite set of processes that can communicate via asynchronous messages. At the higher level, basic MSCs are composed using High-level MSCs (HMSCs for short), a finite automaton labeled by basic MSCs. The language is then very simple and needs only to learn few concepts to be able to model behaviors of distributed systems. It has met considerable interest, both from an academic and industrial point of view. Indeed, this model has raised new theoretical problems (verification, synthesis, ...) but also renewed the interest for existing problems and solutions in other concurrency models such as traces [15]. From an industrial point of view, MSCs or their UML variant (sequence diagrams) have also been used to model requirements in distributed systems [4], and have proved to be efficient modeling tools to discover errors at early stages of system design. HMSCs are very expressive. The usual verification techniques that usually apply to finite state machines (model checking of temporal logic formulae, intersecting two specifications) are in general undecidable for HMSCs. Fortunately, several syntactic classes of HMSCs with more decidable problems have been characterized [3, 6, 9]. The restriction to these classes should however remain a reasonable limitation of the expressive power of HMSCs that does not impact the usability of the model. Restricting to bounded HMSCs [3], for instance, reduces the expressive power of HMSCs to that of finite automata, which is in general too restrictive for the kind of asynchronous applications that are usually modeled with scenarios. The other difficulty is that in general, HMSCs are not implementable. It means that system designers can not synthesize a distributed program from HMSCs, and hence can not reuse all the modeling and verification work performed on HMSCs to guarantee correctness of the synthesized implementation.

Again, designers can rely on syntactic subclasses of HMSCs that can be implemented. However, proposed classes until now are not general enough. The *reconstructible* HMSCs of [11] impose restrictions on the way loops and choices are used, and the solution in [9] imposes conditions on the set of active processes in a basic MSC. Other implementation techniques implement exactly the language of a bounded HMSC, but with deadlocks [14], or avoid deadlocks but need several initial states in the synthesized machines [5]. In our opinion, deadlocks should not be allowed in an implementation of an HMSC : it amounts to deciding at runtime that an ongoing execution is not valid, as it does not belong to the specification. In this work, we consider that a realistic implementation of HMSCs should not allow deadlocks, at least in distributed systems that do not allow compensation and backtracking. We furthermore consider that a distributed system implements an HMSC if the prefix closure of the behaviors of the original specification and of the synthesized machines are identical.

This paper proposes an implementation mechanism for local message sequence charts, that is HMSC specifications that do not require distributed consensus to be implementable. The proposed technique is to project an HMSC on each process participating to the specification. It is well known that this solution produces programs with more behaviors than in the specification [11]. We then compose these projections with local controllers, that intercept mes-

sages between processes and tag them with sufficient information to avoid the additional behaviors that appear in the sole projection. The main result of this work is that the projection of the behavior of the controlled system on events of the original processes is equivalent (up to a renaming) to the behavior of the original HMSC.

This paper is organized as follows. Section 2 defines the formal models that will be used in the next sections. Section 3 characterizes an interesting syntactic class of HMSCs called local HMSCs. Section 4 defines the projection operation, that generates communicating finite state machines from an HMSC, and shows that an HMSC and its projection are not equivalent in general. Section 5 proposes a solution based on local control and message tagging to implement properly an HMSC. Finally, section 6 presents a small prototype before conclusion.

2 Definitions

MSCs is a partial-order based standard formalism. It is a visual notation, which clearly expresses interactions among concurrent processes. It consists essentially of a finite set of processes (also called instances) denoted I, that run in parallel and exchange messages in a one-to-one, asynchronous fashion. MSCs and their variants are widely used to capture use cases and requirements during the early design stages of distributed systems. They have been adopted within several software engineering methodologies and tools for concurrent, reactive and real-time systems. e.g. [1, 2, 8, 16].

A basic MSC is a diagram that defines a simple communication scenario between instances. The life lines of instances are drawn vertically from top to bottom and define sequences of events. An event can be a message sending or reception, or a local action. Horizontal arrows represent asynchronous messages from one instance to another. Examples of bMSCs are shown in Figure 1.

Definition 1 (bMSCs) *A* bMSC *is a tuple* $M = (E, \leq, C, \phi, t, m)$ *where :*

- E *is a finite set of events.*
- $\phi : E \longrightarrow I$ *localizes events on the different instances.* E *can be split into a disjoint union* $\uplus_{p \in I} E_p$*, where* $E_p = \{e \in E \mid \phi(e) = p\}$ *is the set of events occurring on instance* p*.* E *can also be considered as the disjoint union* $S \uplus R \uplus L$ *in order to distinguish send events* ($e \in S$)*, receive events* ($e \in R$) *or local actions* ($e \in L$)*.*
- C *is a finite set of message contents and action names.*
- $t : E \longrightarrow \Sigma$ *gives a type to each event, with* $\Sigma = \{p!q(a), p?q(a), a \mid p, q \in I, a \in C\}$*. We have* $t(e) = p!q(a)$ *if* $e \in E_p \cap S$ *is a send event of message* a *from* p *to* q*,* $t(e) = p?q(a)$ *if* $e \in E_p \cap R$ *is a receive event of message* a *by* p *from* q *and* $t(e) = a$ *if* $e \in E_p \cap L$ *is a local action, named* a *located on* p*.*
- $m : S \longrightarrow R$ *is a bijection that matches send and receive events. If* $m(e) = f$*, then* $t(e) = p!q(a)$ *and* $t(f) = q?p(a)$ *for some* $p, q \in I$ *and* $a \in C$*.*
- $\leq \subseteq E^2$ *is a partial order relation (the "causal order"). It is required that events of the same instance are totally ordered :* $\forall (e_1, e_2) \in E^2$ $\phi(e_1) = \phi(e_2)$ *implies that* $(e_1 \leq e_2) \vee (e_2 \leq e_1)$*. For an instance* p*, let us call* $\leq_p$

this total order. $\leq$ *must also reflect the causality induced by the message exchanges, i.e.* $\leq = (\bigcup_{p \in I} \leq_p \cup m)^*$

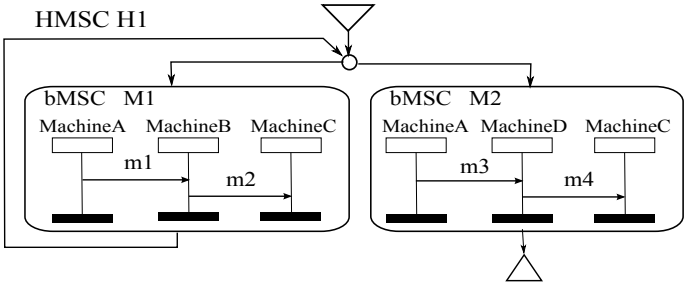

FIGURE 1 – An example of High-level Message Sequence Chart.

For a bMSC M, we will denote by
$min(M) = \{e \in E \mid \forall e' \in E,\ e' \leq e \Rightarrow e' = e\}$, the set of minimal events of M, and by $min_p(M)$ the first event located on instance $p \in I$ (if it exists). An instance is called *minimal* if it carries a minimal event. Similarly, we will denote by $max(M) = \{e \in E \mid \forall e' \in E,\ e \leq e' \Rightarrow e' = e\}$ the set of maximal events of M.

Standard notation of bMSCs also allows for the definition of a zone on an instance axis called *co-region*. It describes the absence of ordering along an instance axis. MSCs also allow behaviors with *message overtaking*, i.e. in which messages are not received in the order of their emission. Without loss of generality, we do not consider these aspects : co-regions can be simulated by adding a finite number of alternatives in an HMSC containing the interleaving of events within coregions, and message crossing can not occur within the targeted architecture. We then suppose that all bMSCs are FIFO, that is for two sending events e, e' such that $p = \varphi(e) = \varphi(e')$, $q = \varphi(m(e)) = \varphi(m(e'))$ we always have $e \leq_p e' \iff m(e) \leq_q m(e')$.

Definition 2 *Let* $M = (E, \leq, C, \phi, t, m)$ *be a bMSC. A* prefix *of* M *is a tuple* $(E', \leq', C', \phi', t', m')$ *such that* E' *is a subset of* E *closed by causal precedence (i.e.* $e \in E'$ *and* $f \leq e$ *implies* $f \in E'$*) and* $\leq', C', \phi', t', m'$ *are restrictions of* $\leq, C, \phi, t, m$ *to* E'*. A* suffix *of* M *is a tuple* $(E', \leq', C', \phi', t', m')$ *such that* E' *is closed by causal succession (i.e.* $e \in E'$ *and* $e \leq f$ *implies* $f \in E'$*) and* $\leq', C', \phi', t', m'$ *are restrictions of* $\leq, C, \phi, t, m$ *to* E'*. A* piece *of* M *is the restriction of* M *to a set of events* $E' = E \setminus X \setminus Y$*, such that the restriction of* M *to* X *is a prefix of* M *and the restriction of* M *to* Y *is a suffix of* M*.*

Note that prefixes, suffixes and pieces are not always bMSCs, as the message mapping m is not necessarily a bijection from sending events to receiving events. In the rest of the paper, we will denote by $Pref(M)$ the set of all prefixes of a bMSC M.

Definition 3 (Sequencing) *The sequencing operator $\circ$ for two bMSCs $M_1 = (E_1, \leq_1, C_1, \phi_1, t_1, m_1)$ and $M_2 = (E_2, \leq_2, C_2, \phi_2, t_2, m_2)$ consists in concatenation of the two bMSCs instance by instance. $M_1 \circ M_2 = (E, \leq, C, \phi, t, m)$, where :*

- *$E = \varphi_1(E_1) \uplus \varphi_2(E_2)$ where φ_1 and φ_2 are two isomorphisms with disjoint images.*
- *$\forall e, e' \in E$, $e \leq e'$ iff $\varphi_1^{-1}(e) \leq_1 \varphi_1^{-1}(e')$ or $\varphi_2^{-1}(e) \leq_2 \varphi_2^{-1}(e')$ or $\exists (e_1, e_2) \in E_1 \times E_2 : \phi_1(e_1) = \phi_2(e_2) \wedge \varphi_1^{-1}(e) \leq_1 e_1 \wedge e_2 \leq_2 \varphi_2^{-1}(e')$*
- *$\forall e \in E, \phi(e) = \phi_1(\varphi_1^{-1}(e))$ if $e \in \varphi_1(E_1)$ or $\phi(e) = \phi_2(\varphi_2^{-1}(e))$ if $e \in \varphi_2(E_2)$ and $m(e) = m_1(\varphi_1^{-1}(e))$ if $e \in \varphi_1(E_1)$ or $m(e) = m_2(\varphi_2^{-1}(e))$ if $e \in \varphi_2(E_2)$.*

An example of concatenation is shown in Figure 2. In the next sections, we will also need to concatenate prefixes and pieces of bMSCs. Prefix and piece concatenation is defined alike bMSC concatenation with an additional phase that rebuilds the message mappings. Let O_1 be a prefix of a bMSC, and O_2 be a piece of MSC. Then, the concatenation of O_1 and O_2 is denoted by $O_1 \circ O_2 = (E, \leq, C, \phi, t, m)$, where $E, \leq, C, \phi$, and t are defined as in definition 3 and m is a function that associates the n^{th} sending event from p to q to the n^{th} reception from p on q for every pair of processes $p, q \in I$. Note that this sequencing is not defined if for some p, q, n, the types of the n^{th} sending and reception do not match, that is one event is of the form $p!q(m)$ and the other one $q?p(n)$ with $n \neq m$. In particular, we will denote by $O \circ \{a\}$ the prefix obtained by concatenation of a single event a to a prefix O.

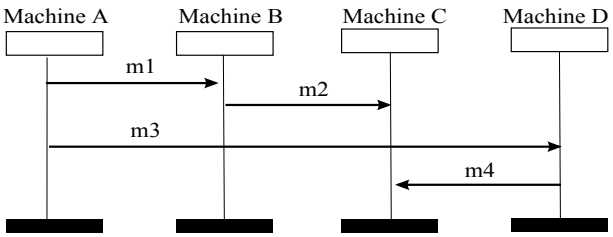

FIGURE 2 – The bMSC obtained by concatenation of M_1 and M_2 in Fig. 1.

HMSC diagrams are automata that compose basic MSCs or other HMSCs. The whole language contains iteration, choice, and parallel composition. In this paper, we restrict to HMSCs without parallel frames, and suppose without loss of generality that our HMSCs comprise only one hierarchical level, i.e. they are automata labeled by bMSCs.

Definition 4 (HMSCs) *A* HMSC *is a graph $H = (I, N, \rightarrow, \mathcal{M}, n_0)$, where*

- *I is a finite set of instances.*
- *N is a finite set of nodes and $n_0 \in N$ is the initial node of H.*
- *$\mathcal{M}$ is a finite set of bMSCs which participating instances belong to I, and defined on disjoint sets of events.*
- *$\rightarrow \subseteq N \times \mathcal{M} \times N$ is the transition relation.*

Figure 1 shows a complete HMSC. In the rest of the paper, and without loss of generality, we will consider that all nodes, except possibly the initial node and

the sink nodes are choice nodes, i.e. have several successors by the transition relation (by finite concatenation of bMSCs, an HMSC can be always transformed in such a canonical form). We also require HMSCs to be deterministic (this can be ensured by the standard determinization procedure of finite automata).

Definition 5 (HMSC behavior) *As usual, for an HMSC H, we define as $Paths(H)$ the set of paths of H starting from the initial node. We will say that a path is* acyclic *if and only if it does not contain twice the same transition. A path $\rho \in Paths(H)$ defines a sequence of $\mathcal{M}^*$ and thus the corresponding bMSC formed by concatenation and denoted by O_ρ. Let us denote by $\mathcal{L}(H) = \bigcup_{\rho \in Paths(H)} Pref(O_\rho)$, the set of behaviors of the HMSC H.*

Note that our definition of the language of an HMSC H includes all prefixes of bMSCs generated by H. A *correct implementation* of an HMSC H is a distributed system which reproduces exactly (and nothing more) $\mathcal{L}(H)$.

3 Local HMSCs

A bMSC can have several minimal instances, called *deciding instances*. The set of deciding instances in a bMSC M is simply $\phi(Min(M))$. They carry the first events that happen in M. Obviously, these events cannot be message receptions. So the deciding instances can choose to perform exactly the same scenario at a choice node. The other non-deciding instances have to conform to the chosen scenario. This raises a problem : if the outgoing transitions of a choice node are labeled by bMSCs with distinct deciding instances, then, without synchronization among them in the distributed implementation, one instance might decide to perform one scenario M_1, and another instance an incompatible scenario M_2. There are two solutions to avoid this bad situation : either impose synchronization of all processes before each choice with more than one deciding instance to elect the next scenario to perform, or restrict to HMSCs in which a single instance decides at each choice (the so-called local-choice HMSCs [6]). The first solution has the drawback that it adds synchronization and consensus mechanisms to the implementation, and hence modifies the specification. We will show later in the paper that working with local-choice HMSCs does not always guarantee that the implementation is correct.

Definition 6 (Local choice node) *Let $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ be an HMSC, and let $c \in N$. Choice c is* local *if and only if for every pair of (non necessarily distinct) paths $\rho = c \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \dots n_k$ and $\rho' = c \xrightarrow{M'_1} n_1 \xrightarrow{M'_2} n'_2 \dots n'_k$ there is a single minimal instance in O_ρ and in $O_{\rho'}$ (i.e. $\phi(Min(O_\rho)) = \phi(Min(O'_\rho))$ and $|\phi(Min(O_\rho))| = 1$). H is called a* local-choice *HMSC if all its choices are local.*

Intuitively, the local-choice property [6] guarantees that every choice is controlled by a unique instance.

Theorem 1 (Deciding locality) *Let H be an HMSC. H is* not local *iff there exists a node c and a pair of* ***acyclic*** *paths ρ, ρ' originating from c, such that O_ρ and $O_{\rho'}$ have more than one minimal instance.*

Proof : one direction is straightforward : if we can find a node c and two (acyclic) paths with more than one deciding instance, then obviously, c is not a local choice, and H is not local. Let us suppose now that for every node c, and for every pair of acyclic paths of H originating from c, we have only one deciding instance. Now, let us suppose that there exist a node c_1 and two paths ρ_1, ρ'_1 such that at least one (say ρ_1) of them is not acyclic. Then ρ_1 has a finite acyclic prefix w_1. The set of minimal instances in O_{w1} and in $O_{\rho 1}$ is the same, as $\phi(min(M \circ M)) = \phi(min(M))$. Hence, c, ρ_1, ρ'_1 are witnesses for the non-locality of H iff c, w_1, ρ'_1 are also such witnesses. □

Theorem 2 (Complexity of local choice) *Deciding if an HMSC is local-choice is in* $co - NP$.

Proof : the objective is to find a counter example, that is two paths originating from the same node with distinct deciding instances. One can choose in linear time in the size of H a node c and two finite acyclic paths ρ_1, ρ_2 of H starting from c, that is sequences of MSCs of the form $M_1 \dots M_k$. One can compute a concatenation $O = M_1 \circ \dots \circ M_k$ in polynomial time in the total size of the ordering relations. Note that to compute minimal events of a sequencing of two bMSCs, one does not have to compute the whole causal ordering $\leq$, and only has to ensure that maximal and minimal events on each instance in two concatenated MSCs are ordered in the resulting concatenation. Hence it is sufficient to recall a covering of the local ordering $\leq_p$ on each process $p \in I$ plus the message relation m. Then finding the minimal events (or equivalently the minimal instances) of O can also be performed in polynomial time in the number of events of O, as $Min(M) = E \setminus \{f \mid \exists e, e \leq_p f \vee f = m(e)\}$. □

From theorem 1, an algorithm that checks locality of HMSCs is straightforward. It consists in a width first traversal of acyclic paths starting from each node of the HMSC. If at some time we find two paths with more than one minimal instance, then the choice from which these path starts is not local. Note that minimal instances need not be computed for the whole MSC labeling each path, and can be updated at the same time as paths. Indeed, if $\rho = \rho_1.\rho_2$ is a path of H, then $\phi(Min(M_\rho)) = \phi(Min(M_{\rho_1})) \cup (\phi(Min(M_{\rho_2})) \setminus \phi(M_{\rho_1}))$. It is then sufficient for each path to maintain the set of instances that appear along this path, and the set of minimal instances, without memorizing exactly the scenario played. As we consider only acyclic paths of the HMSC the following algorithm terminates.

The following algorithm was proposed in [11]. It builds a set of acyclic paths starting from each node of an HMSC. A non-local choice is detected if there is more than one deciding instance for a node c. The algorithm remembers a set of acyclic paths P, extends all of its members with new transitions when possible, and places a path ρ in MAP as soon as the set of transitions used in ρ contains a cycle.

Algorithm 1 LocalChoice(H)

for c node of H **do**
  P $= \{(t, I, J) \mid t = (c, M, n) \wedge I = \phi(min(M)) \wedge J = \phi(M)\}$
  MAP $= \emptyset$ ***/*Maximal acyclic Paths*/***
  while $P \neq \emptyset$ **do**
    MAP $=$ MAP $\cup$ $\{(w.t, I') \mid w = t_1...t_k \wedge t_k = (n_{k-1}, M_k, n_k), t = (n_k, M, n) \wedge t \in w \wedge (w, I, J) \in P \wedge I' = I \cup (\phi(min(M)) - J)\}$
    P $= \{(w.t, I', J') \mid (w, I, J) \in P, w = n_1...n_k \wedge t_k = (n_{k-1}, M_k, n_k), t = (n_k, M, n) \wedge t \notin w \wedge J' = J \cup \phi(M) \wedge I' = I \cup (\phi(min(M)) - J)\}$
  end while
  DI $= \bigcup_{(w,I) \in MAP}$ I ***/*Deciding Instances*/***
  if $\mid DI \mid > 1$ **then**
    H contains a non-local choice c
  end if
end for

4 The synthesis Problem

In this section, we introduce our implementation model, namely Communicating Finite State Machines (CFSM) [7]. A CFSM $\mathcal{A}$ is a network of finite state machines that communicate over unbounded, non-lossy, error-free and FIFO communication channels. We will write $\mathcal{A} = \parallel_{i \in I} A_i$ to denote that $\mathcal{A}$ is a network of machines describing the behaviors of a set of machines $\{A_i\}_{i \in I}$. A communication buffer $B_{(i,j)}$ is associated to each pair of instances $(p, q) \in I^2$. Buffers will implement messages exchanges defined in the original HMSC. Now that HMSCs (a global specification) and communicating finite state machines (a set of local reactive machines, without global control) are defined, we can describe the synthesis problem as follows : given an HMSC H over a set of instances I, build a set of communicating state machines $A_1, \ldots A_{|I|}$ which behaves exactly as $\mathcal{L}(H)$.

An obvious solution is to project the original HMSC on each instance. Unfortunately, this solution does not work for all HMSCs. Consider for instance a non local HMSC that contains a choice c, such that $c \xrightarrow{M_1} n_1$ and $c \xrightarrow{M_2} n_2$, and M_1 and M_2 have two distinct deciding instances p and q. Then, the projection of H on its instances produces automata that can behave exactly as in H up to choice c, and then p decides to execute scenario M_1 while q decides to execute scenario M_2. Clearly, this execution was not defined in H. Hence, the projection of an HMSC on its instances can define more behaviors than the original specification, but can also deadlock. In the rest of the paper, we will only consider local-choice HMSCs. However, we will show in this section that this is not sufficient to ensure correctness. We will also consider that two MSC labeling distinct transitions of a local HMSC start with distinct messages. This assumption will be used to differentiate distinct choices at runtime. It is not essential in our framework, as we may enforce instances to tag their messages with the transition of the HMSC currently executed by the instance. However, it greatly simplifies the notations and proofs.

The principle of projection is to copy the original HMSC on each instance, and to remove all the events that do not belong to the considered instance. By construction, we keep the structure of the HMSC automaton. Each transition is a sequence (possibly empty) of events. This object can be considered as a finite state automaton by adding intermediary states in sequences of events. Empty transitions can be removed by the usual ε-closure procedure for finite state automata.

Definition 7 (Projection) *Let us consider an HMSC* $H = (I, N, \rightarrow, \mathcal{M}, n_0)$. *The set of events of a bMSC* M *is denoted by* E_M, *and the set of events of* M *located on instance* i *by* E_{Mi}. *States can be coded by tuples* $Q = N \times \mathcal{M} \times N \times \mathbb{N}$, *the first three components designating an HMSC transition and the last one designating the rank of the event on the considered instance in the bMSC labeling this transition. For an instance* $i \in I$, *we define the finite state automaton* A_i, *result of the projection of* H *onto the instance* i *by* $A_i = (Q, \rightarrow_i, E_i \cup \{\varepsilon\}, n_0)$ *with* $E_i = \bigcup_{M \in \mathcal{M}} E_{Mi}$. *The set* E_{Mi} *is totally ordered. We denote its elements by* $e_1, \cdots, e_{|E_{Mi}|}$. *With the convention that* $(n, M, n', 0) \equiv n$ *and* $(n, M, n', |E_{Mi}|) \equiv n'$, $\rightarrow_i$ *is the smallest set such that* $\forall n, n' \in N, \forall M \in \mathcal{M}$, *if* $E_{Mi} = \emptyset$, $(n, \varepsilon, n') \in \rightarrow_i$, *else* $\bigcup_{1 \leq k \leq |E_{Mi}|}((n, M, n', k-1), e_k, (n, M, n', k)) \in \rightarrow_i$.

Each run of this set of communicating machines defines a prefix, that can be built incrementally starting from the empty prefix, and appending one executed event after the other (i.e. it is built from a total ordering of all events occurring on the same process, plus a pairing of messages sending and receptions). From this definition, the language $\mathcal{L}(\mathcal{A})$ of a set of communicating machines is the set of all prefixes associated to runs of $\mathcal{A}$.

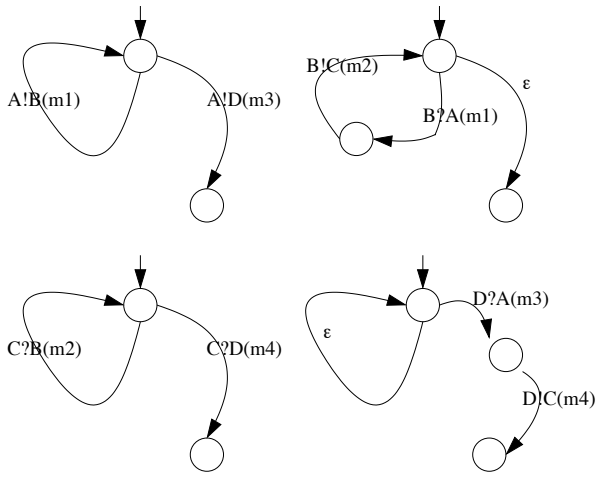

FIGURE 3 – The instance automata projected from the HMSC of Fig. 1.

Let us consider the projections of H in Figure 1 on all its instances given in Figure 3. A correct behavior of H is shown in the left part of Figure 4, while a possible but incorrect behavior of the distributed interaction of the projected

machines is shown in the right part. We can see that message m_3 sent by machine A to machine D can be delayed and received after message m_4 from machine D, hence mixing two different basic MSCs. Machine C does not have enough information to decide to delay the reception of m_4 according to the HMSC semantics.

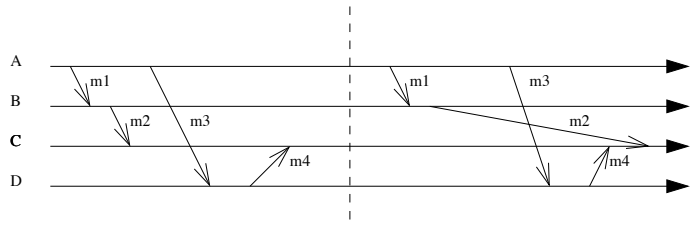

FIGURE 4 – A correct behavior of the HMSC of Fig. 1, and a possible distortion due to the distributed interaction of the projected instances.

This example proves that in general, even for local HMSCs, the synthesis by projection is not correct. It is however proved [11] that the set of behaviors of the synthesized specification contains the set of behaviors of the original specification. The machines synthesized by projection allow more behaviors than the original global specification because some global ordering among actions is lost during projection. A subclass of local HMSCs called *reconstructible* HMSCs that avoids this problem has been proposed [11]. This reconstructibility condition says that local ordering among events from two consecutive choices originating from the same node can be deduced from the FIFO property and from the ordering among minimal events in each chosen MSC. However, reconstructibility is easily lost when two instances communicate indirectly through distinct sets of instances, as in the example of figure 1. Another solution proposed by [9] restricts to a class of local HMSCs in which all processes take part in every branch. Both approaches are restrictive, as they constrain the way processes are used in each branch of an HMSC.

In the next section, we take a different approach, and show that adding information to messages is sufficient to obtain a correct synthesis in the general case of local HMSCs.

5 Implementing HMSCs with message Controllers

We propose a new implementation mechanism, that will allow for the implementation of any local HMSC H, without syntactic restriction. The architecture is as follows : for each process, we compute an automaton, as shown in previous section by projection of H on each of its instances. The projection is the same as previously, with the slight difference that the synthesized automaton communicates with his controller, and not directly with other processes. To differentiate, we will denote by $K(A_i)$ the "controlled version" of A_i, keeping in mind that A_i and $K(A_i)$ are isomorphic machines. Then, we add to each automaton $K(A_i)$ a controller C_i, that will receive all communications from $K(A_i)$, and tag them

with a stamp. Similarly, each controller will receive all tagged messages destinated to $K(A_i)$, and decide with respect to its tag whether a message must be sent immediately to $K(A_i)$ or delayed. Automata and their controller communicate via FIFO channels, which defines a total ordering on message receptions or sendings. In this section, we first define the distributed architecture and the tagging mechanism that will allow for preservation of the global specification. We then define control automata and their composition with synthesized automata. We then show that for local HMSCs the controlled local system obtained by projection behaves exactly as the global specification (up to some renaming and projection that hides the controllers).

5.1 Distributed architecture

We consider the $n = |I|$ automata $\{K(A_i)\}_{1 \leq i \leq n}$ obtained by projection of the original HMSC on the different instances. These automata $K(A_i)$ are connected via a bidirectional FIFO channel (port P) to their associated controller C_i (port P_i). The controllers are themselves interconnected via a complete graph of bidirectional FIFO channels (for all $i \neq j$, port P_j of controller C_i is connected to the port P_i of controller C_j). This is illustrated in Figure 5. This architecture is quite flexible : all the components run asynchronously and exchange messages, without any other assumption on the way they share resources, memory or processors.

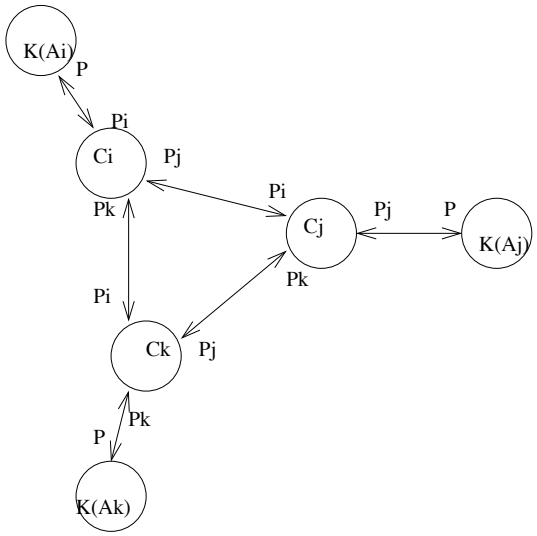

FIGURE 5 – The distributed controlled architecture.

5.2 Tagging mechanism

Definition 8 *Let H be a local HMSC. We set an ordering $\lhd$ on all nodes of H, and adopt a lexicographical ordering $\lhd_l$ on labels of Σ. A* branch *of H is a transition (c, M, n) of H, such that c is a choice node. We will denote by $\mathcal{B}_H$ all branches of H. For a given choice node c, we will suppose that all branches*

stemming from c are pairwise distinguishable from their minimal actions, that is if (c, M, n) *and* (c, M', n') *are two branches, then* $t(min(M)) \neq t(min(M'))$. *The ordering on nodes extends to branches : we will say that* $b_1 = (c_1, M_1, n_1)$ *precedes* $b_2 = (c_2, M_2, n_2)$ *and write*

$$b_1 \lhd b_2 \text{ iff } \begin{cases} c_1 \lhd c_2, & \text{or} \\ c_1 = c_2 & \text{and } n_1 \lhd n_2, \text{ or} \\ c_1 = c_2 & \text{and } n_1 = n_2 \\ & \text{and } t(min(M_1)) \lhd_l t(min(M_2)) \end{cases}$$

Definition 9 (Choice clocks) *A* choice clock *of an HMSC H is a vector of* $\mathbb{N}^{\mathcal{B}_H}$. *Let* $\rho = n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \ldots \xrightarrow{M_k} n_k$ *be a path of H. The* choice clocks labeling *of* O_ρ *is a mapping* $\tau : E_{O_\rho} \longrightarrow \mathbb{N}^{\mathcal{B}_H}$ *such that for every* $i \in 1..k, e \in M_i$, $\tau(e)[b]$ *is the number of occurrences of branch b in* $n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \ldots \xrightarrow{M_i} n_i$

The usual terminology and definitions on vectors apply to choice clocks. A vector V_2 is an *immediate successor* of a vector V_1 of same size, denoted $V_1 \lessdot V_2$, if there is a single component b such that $V_1[b]+1 = V_2[b]$, and $V_1[b'] = V_2[b']$ for all other entries b'. Vectors V_1 and V_2 are equal if $V_1[b] = V_2[b]$ for every entry, and V_2 is greater than V_1 iff $V_1[b] = V_2[b]$ for some entries b, and $V_1[b] < V_2[b]$ for all others.

For a given path $\rho = n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \ldots \xrightarrow{M_k} n_k$, we will call the *choice events* of O_ρ the minimal events in every $M_i, i \in 1..k$. It is rather straightforward to see that when an HMSC H is local, then for every path ρ of H, the set of choice events in O_ρ is totally ordered. Note also that for a pair of events e, f in O_ρ, $\tau(e) = \tau(f)$ if and only if e, f belong to the same MSC M_i. From these facts, the following proposition is straightforward :

Proposition 1 *Let H be a local HMSC, and* ρ *be a path of H. Let* $\prec$ *be the usual ordering on integer vectors. Then* $(\tau(E_{O_\rho}), \prec)$ *is a totally ordered set.*

This proposition is important : it means that if one is able to maintain locally a consistent tagging of messages, then a local automaton that receives two messages can always decide which message should be received first.

Definition 10 (Concerned instances) *Let* $b = (c, M, n)$ *be a branch of an HMSC H. We will say that instance* $p \in \mathcal{I}$ *is concerned by branch b if and only if there exists an event of M on p* ($E_{M_p} \neq \emptyset$). *Let* $K \in \mathbb{N}^{\mathcal{B}_H}$ *be a choice clock, and let* $p \in \mathcal{I}$ *be an instance of H. The vector of choices that concern p in K is the restriction of K to branches that concern p, and is denoted by* $[K]_p$.

For a given instance i, the controller C_i associated with the projected automaton $K(A_i)$ will receive the messages sent by $K(A_i)$ and by the other controllers. We consider that messages exchanged between the automata and the controllers are triple (j, m, b) where $j \in I$ is the destination automaton, $m \in C$ is the message name, and b the branch in which the sending event has occurred. In other words, in our controlled architecture, an automaton executes $p!C_p(q, m, b)$ instead of $p!q(m)$. The messages exchanged between controllers are tagged and represented by pairs (m, τ) where m is a message name and $\tau \in \mathbb{N}^{\mathcal{B}_H}$ a choice vector. In addition, the controller C_i maintains several local variables :

- $\tau_i \in \mathbb{N}^{\mathcal{B}_H}$, its local known choices vector. It is initialized to the null vector, and updated upon consumption of incoming messages.
- *nbevt*, which counts the remaining number of communication events of the instance i to be treated in the current branch that is being processed.
- *Rec* is a sequence of reception events. *nbevt* and *Rec* are initialized with constant values (that depend on the chosen branch) when dealing with the first event of a branch on process i.
- *currentb*, which memorizes the branch of H that is currently executed by process i.

In the rest of the paper, we will denote by $\pi_i(M)$ the sequence of events obtained by projection of M on instance $i \in \mathcal{I}$, and by $\pi_{i,?}(M)$ the restriction of this sequence to receptions. The generic algorithm for a controller C_i is composed of two rules, which are continuously active. Rule 1 applies to communications from $K(A_i)$ to C_i. First case corresponds to minimal events controlled by the projected automaton $K(A_i)$. When dealing with the first event of the bMSC (branch b) to be processed, the only role of the controller is to compute the tag (increment of the corresponding component of τ_i) and to initialize the variables *nbevt* and *Rec*. The currently processed branch is stored in variable *currentb*. The other case deals with communications from $K(A_i)$ that are not choices of $K(A_i)$. These events are generated in correct order by construction of the projection.

The second rule applies for every port $P_j, j \neq i$, and aims at controlling the order of the different receptions. This is the rationale of the controller. There are three cases. The first case occurs when a branch of H has already been started, that is a controller C_i has received (i.e. consumed) a message indicating the choice performed by the deciding instance of this branch, and a valid message arrives. In this situation, all the components concerning $K(A_i)$ of the current tag τ_i and of the tag τ labeling the incoming message must be equal, and this incoming message must be the next expected message (i.e. the next reception in *Rec*) in currently executed branch. Then the message can be consumed by C_i and forwarded to $K(A_i)$. The fact that there is only one FIFO channel between the controller C_i and the projected automaton $K(A_i)$ ensures the correct order of receptions on this automaton. The second case is when the incoming message is the first communication signaling a new choice. The controller then checks if the received message defines the next branch of H that must be executed by $K(A_i)$. This is done by verifying if the received tag is the next tag to be treated (considering only the components that concern $K(A_i)$), that is $[\tau_i]_i \lessdot [\tau]_i$. In that case, the current tag can be updated. The current branch is retrieved by considering the component that differs between $[\tau]_i$ and $[\tau_i]_i$. Then the remaining number of events that should be executed within this branch (the number of events on the instance i in the bMSC of the current branch, minored by one) is set, as well as the expected sequence of receptions, before transmission of the message to $K(A_i)$. The third case applies when none of the above situations hold, that is the incoming message on port P_j can not yet be consumed, either because it is not the next reception expected (another reception on another port should occur before this one) or the incoming message signals that a new choice has been started, but more events must occur before consuming it. In such case, the controller does nothing, and waits for other messages on other ports.

Algorithm 2 Controller C_i

RULE 1 : when (j, m, b) available on port P_i
consume (j, m, b)
if $nbevt = 0$ **then**
  $\tau_i[b]$++
  $nbevt = |\Pi_i(M_b)| - 1$
  $Rec = \Pi_{i,?}(M_b)$
  send (m, τ_i) on port P_j
else
  $nbevt$ - -
  send (m, τ_i) on port P_j
end if

RULE 2 : when (m, τ) available on port P_j
if $([\tau_i]_i = [\tau]_i) \wedge (Rec = A_i?A_j(m).w)$ **then**
  consume (m, τ)
  $nbevt$ - -
  send (j, m) on port P_i
  $Rec = w$
else
  if $(nbevt = 0) \wedge ([\tau_i]_i \lessdot [\tau]_i)$ **then**
    consume (m, τ)
    $\tau_i = \tau$
    $currentb = b$ s.t. $[\tau][b] - [\tau_i][b] \neq 0$
    $nbevt = |\Pi_i(M_{currentb})| - 1$
    $Rec = \Pi_{i,?}(M_{currentb})_{[2..nbevt]}$
    send (j, m) on port P_i
  end if
end if

5.3 Proof of correctness

Definition 11 *Let $O = (E, \leq, t, \phi, m)$ be a prefix in $\mathcal{L}(||K(A_i)|Ci)$. The* restriction *of O to non-control events is a restriction of O to events located on $K(A_i)$'s. We will denote by $Unc(O)$ this restriction. The* uncontroling *of $O = (E, \leq, t, \phi, m)$ is a renaming function $Ru()$ that replaces communications to and from the controller of a process by direct communications with the process concerned by the sent/received message, and builds the message mapping. $Ru(O) = (E, \leq, t', \phi, m')$, where $t'(e) = p!q(m)$ if $t(e) = p!C_p(m, q, c)$, $t'(e) = p?q(m)$ if $t(e) = p?C_p(m, q)$, and $t'(e) = t(e)$ otherwise. Function m' maps the i^{th} sending from p to q with the i^{th} reception on q from p for every pair of processes.*

Note that for a prefix O in $\mathcal{L}(||K(A_i)|Ci)$, the message mapping in $Unc(O)$ is an empty relation.

Theorem 3 *Let H be an HMSC, and let $||K(A_i)|C_i$ be its controlled synthesis. Then, $Ru(Unc(\mathcal{L}(||K(A_i)|C_i))) = \mathcal{L}(H)$.*

Proof : We will prove this theorem by showing inclusion in the two directions. We will also need a technical lemma that shows that the tagging mechanism is the same in an HMSC and in its implementation.

Lemma 1 *For all H, local choice HMSC, the choices events in any behavior of the synthesized communicating machines are totally ordered.*

proof : We will prove this property by induction. Let us denote by P_n the property : for all H, local choice MSC, the choices in any behavior of the synthesized communicating machines in a run containing n choices are totally ordered.

Let us first verify this property for $n = 2$. As H is local choice, then there is only one CFM that can perform an action (a message sending) from the initial configuration. The next choice can then only be played after the first one. Hence, the first two choices are ordered.

Let us suppose the property verified up to n, and prove that it also holds for $n+1$. Let us suppose a prefix $O \circ O'$ from $\mathcal{L}(||K(A_i)|C_i)$ with $n+1$ choices, such that O contains n choices. Then, O is of the form $O = \{c_1\} \circ O_1 \ldots \{c_n\} \circ O_n$, where each c_i is a choice event, and such that $\{c_1\} \circ O_1$ is a prefix of the first MSC M_1 played in this run. Then, $O \circ O'$ can be completed by piece P_1 such that $O \circ O' \circ P_1$ contains a complete execution of the first MSC M_1 (so far, nothing forces M_1 to be completely executed). $O \circ O' \circ P_1$ can be equivalently rewritten as $O \circ O_{1,1} \circ \ldots O_{1,n} \circ P_1 \circ \{c_2\} \circ O'_2 \ldots \{c_n\} \circ O'_n \circ O'$, where each $O_{1,i}$ is the part of O_i that belongs to M_1 and $O'_i = O_i \setminus O_{1,i}$, as events in $\{c_2\} \ldots \{c_n\} \circ O'_n \circ O'$ do not need to wait for the execution of any event in P_1 to be fireable. Note that P_1 is ensured to be a legal continuation of $O \circ O'$ as no machine can start executing events with tag greater than $0^{\mathcal{B}_H}$ before executing its task in M_1. Let us denote by $P_{2,n} = \{c_2\} \ldots \{c_n\} \circ O'_n \circ O'$ the tagged piece starting at choice event c_2, and by $P'_{2,n}$ the same piece, where all tags are decremented on component M_1. $P'_{2,n}$ is a run with n choices of an HMSC H', which is a copy of H where the initial node is the node reached in H after M_1. Hence, all choices in $P'_{2,n}$ are ordered, and so are choices in w'. Hence, all choices in $O \circ O'$ are totally ordered.□

As choice events are the only moment when a tag is updated, this lemma also means that the set of tags that can appear in an execution is the set of tags labeling choice events, and hence that the tags produced in any run that belongs both to $\mathcal{L}(H)$ and $\mathcal{L}(||K(A_i)|C_i)$ are the same. We are now ready to prove the two inclusions.

I) First let us show that $Ru(Unc(\mathcal{L}(||K(A_i)|C_i))) \subseteq \mathcal{L}(H)$ (rewritten for short as $\mathcal{L}_1 \subseteq \mathcal{L}_2$).

Suppose that there exists a prefix $O \circ \{a\} \in \mathcal{L}_1$ such that $O \in \mathcal{L}_2$, but $O \circ \{a\} \notin \mathcal{L}_2$. Suppose that a is a sending event from p to q, i.e. $a = p!q(m)$ for some m. O brings the CFSM in a configuration C_A and H in a configuration C_H. The automaton $K(A_p)$ is in a state from which a can be fired, that is all predecessors of a on p have been executed both in C_A and C_H. Hence, action a is also allowed from C_H, as projection preserves sequences of events on each process. Contradiction. A similar case holds for atomic actions. Hence, a can only be a receive action, i.e. a is of the form $a = p?q(m)$ for some q, m. Then, there exists a prefix $O' \in \mathcal{L}(||K(A_i)|C_i)$ such that $RU(Unc(O')) = O$, and O' is of the form $O_1 \circ \{a'\} \circ .O_2$, where O_1 is a prefix, and O_2 a piece of bMSC. Action a' is of the form $a' = C_p?C_q(m, \tau, b)$, otherwise a can not be played after O. O_1 is of the form $O_1 = P_1.\{b\}.P_2.\{b'\}.P_3$ where $b = K(A_q)!C_q(m, p, b)$ and $b' = C_q!C_p(m, \tau, b)$, with the same tag τ as in a'.

After O, the automata are in a configuration where the FIFO queue from C_p to $K(A_p)$ has a message m as head. In configuration C_H corresponding to O there is a message sent but not received from q to p. If a is not allowed in C_H,

then it is not because the message to be received was not sent, but rather because there exists some former events on process p to be executed (i.e. there is piece of bMSC O_a such that a can be played from C_H only after O_a has been played. Let us get back to the communicating automaton $K(A_p)$. $K(A_p)$ has reached a state s in C_A. As there exists a transition by a from $K(A_p)$ and as we know that there exists also several events in O_a that can be executed from the configuration C_H, then state s is a choice, from which a and the minimal event of O_a on instance p are possible. O_a and a are hence consequences of different choices, and we have that $\tau(O_a) \neq \tau(a)$. From lemma 1 we know that all choices in an execution of the communicating automata are ordered. As all choices are totally ordered, the tags associated to an execution of an HMSC and to an execution of communicating automata are the same. We then have $\tau(O_a) < \tau(a)$. As O_a and a are events of choices that concern p, the communication $C_q!C_p(m, \tau(a))$ that must occur in O before a can not be played as the message received is tagged by a vector which is not the expected successor tag on C_p. We then have a contradiction, and $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

II) Let us now prove the reverse direction : $\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Let us suppose there exists $O \circ \{a\}$ such that $O \circ \{a\} \in \mathcal{L}_2$, $O \in \mathcal{L}_1$ but $O \circ \{a\} \notin \mathcal{L}_1$. If a is a sending of a message or an atomic action, then all preceding events on the same process have been executed in the CFM, and hence a can be fired, as atomic actions and sendings can be fired as soon as the considered process is in a state that enables it, without considering queues contents. Contradiction.

Then, a is a reception $a = p?q(m)$, and the sending $m^{-1}(a)$ from q to p has been executed in O, that is both in H and in the CFM. A message m from p to q is translated in a CFM execution as a sequence of actions : $s_1 = K(A_q)!C_q(m, p, b)$, $r_1 = C_q?K(A_q)(m, p, b)$, $s_2 = C_q!C_p(m, \tau, b)$, $r_2 = C_p?C_q(m, \tau, b)$, $s_3 = C_p!K(A_p)(m)$, $r_3 = K(A_p)?C_p(m)$. As $O \in \mathcal{L}_1$, we know that r_1 has been executed. Similarly, the automaton $K(A_p)$ is in a sate from which r_3 can be fired, because all predecessors of a on process p have been executed by $K(A_p)$. Then, r_3 can not be executed iff r_1, s_2, r_2 or s_3 has not been executed. Similarly, if all communications from $K(A_q)$ to C_q took place in $RU^{-1}(Unc^{-1}(w))$, then r_1 is either fired or fireable, and as a consequence s_2 can also be fired. Let us suppose that C_p is blocked at r_2. Then it means that the message to receive is not properly tagged, or that C_p still have to perform some interactions from previous choices with $K(A_p)$. This can only be receptions of messages from $K(A_p)$, as otherwise, all events in O would not be executed on $K(A_p)$. Hence, C_p can always consume these messages before playing r_2. Now, let us suppose that the tag of the message sent in s_3 is not the expected tag on C_p. Then it means that C_p should observe other tagged messages from a former choice ; The actions to be performed by C_p can only be receptions, and all the message sendings that should have occurred indeed occurred, for the same reasons as before. Hence, all these events can be executed by C_p before executing r_2 and then s_3. Hence the reception can not be blocked in $\mathcal{L}_2$.□

6 A prototype implementation

We have integrated our synthesis method to the SOFAT toolbox, a scenario manipulation environment [10]. SOFAT provides some procedures to decide

whether an HMSC is bounded, local choice, etc. When possible, SOFAT also proposes to animate the HMSC and builds the corresponding chronograms.

The code generated by our synthesis algorithm is written in Promela, the input language of the SPIN model-checker [12]. In Promela, we can easily describe the distributed architecture and the behavior of each component using the notion of Promela process and guarded commands. Using SPIN, it is then possible to simulate the target code, and output the result of a simulation as a bMSC. Spin also allows to model-check some properties (up to some bound on the contents of channels). The text below shows the Promela code generated from the HMSC of Figure 1 and an example of scenario (in a bMSC layout) as produced by the SPIN simulator is shown in Figure 6. We have also model-checked this code to prove that the order of receptions provided by our algorithm is correct on this example.

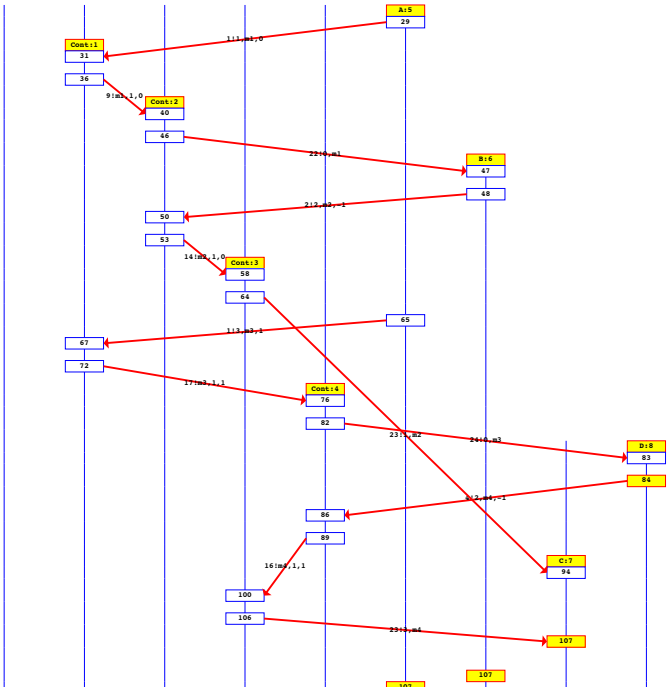

FIGURE 6 – A bMSC produced by SPIN during simulation of the Promela generated code, implementing our synthesis method.

```
/* A Promela implementation of a CFSM generated
   from an HMSC example.
Includes the tagging algorithm -- */

#define CMAX 5 /* max size of channels */
#define NBC 2  /* number of MSCs (choices) */
#define AUTNUM 4 /* the number of automata*/
#define next (((t.T[0]-tau.T[0])*Mask[i].T[0]+\
              (t.T[1]-tau.T[1])*Mask[i].T[1])==1)
#define same (((t.T[0]-tau.T[0])*Mask[i].T[0]+\
              (t.T[1]-tau.T[1])*Mask[i].T[1])==0)
#define update tau.T[0]=t.T[0]; tau.T[1]=t.T[1]
#define diff (t.T[1]-tau.T[1])*Mask[i].T[1]
#define MAXEVTNUM 1 /* the max nb of rec in a bMSC */
#define Shift_Rec /* A macro that shifts the Rec table*/

typedef tagtype { int T[NBC]; }
typedef com_Num { int T[AUTNUM];}
com_Num PI[NBC]; /* number of comm events in a bMSC */
typedef  order_recpt{int recpt[MAXEVTNUM]} /* required seq */
typedef  aut_choice{order_recpt theAUT[AUTNUM]}
aut_choice PIC[NBC];
typedef chan_col {chan PP[AUTNUM]=[CMAX] of {mtype,tagtype};}
chan_col P [AUTNUM]; /* channels between controllers */
chan Aut_Cont[AUTNUM]=[CMAX] of {int,mtype,int};
chan Cont_Aut[AUTNUM]=[CMAX] of {int,mtype};
mtype = {m1,m2,m3,m4} /* message types in the system */
com_Num Mask[AUTNUM]; /* selection of concerned instances */

proctype A()
{sa0: if :: Aut_Cont[0]!1,m1,0; goto sa0
         :: Aut_Cont[0]!3,m3,1 fi}

proctype B()
{sb0: if :: Cont_Aut[1]?0,m1; goto sb1 fi;
 sb1: if :: Aut_Cont[1]!2,m2,-1 fi}

proctype C()
{sc0: if :: Cont_Aut[2]?1,m2; goto sc0
         :: Cont_Aut[2]?3,m4 fi}

proctype D()
{sd0: if :: Cont_Aut[3]?0,m3; goto sd1 fi;
 sd1: if :: Aut_Cont[3]!2,m4,-1 fi}
```

```
proctype Cont(int i) /* The generic controller */
{tagtype tau,t; int nbevt, currentb; int j=0;
 int b; mtype m; order_recpt Rec;
do
/* RULE 1 */ :: Aut_Cont[i]?<j,m,b>;Aut_Cont[i]?j,m,b ->
     if :: (nbevt==0) -> tau.T[b]++; nbevt=PI[b].T[i]-1;
           Rec.recpt[0]=PIC[b].theAUT[i].recpt[0];
           P[j].PP[i]!m,tau
        :: else -> nbevt--; P[j].PP[i]!m,tau fi;
/* RULE 2 */ :: P[i].PP[j]?<m,t> ->
     if :: (same && (Rec.recpt[0]==j)); P[i].PP[j]?m,t ->
           nbevt--; Cont_Aut[i]!j,m ; Shift_Rec
        :: else -> if :: ((nbevt==0) && next);
                P[i].PP[j]?m,t ->
                  update;
                  currentb=diff; nbevt=PI[currentb].T[i]-1;
                  Rec.recpt[0]=PIC[currentb].theAUT[i].recpt[0];
                  Cont_Aut[i]!j,m fi;
               fi;
        :: j=(j+1)%AUTNUM
od
}

init{ /* Constant values obtained from the HMSC parsing */
PI[0].T[0]=1; PI[0].T[1]=2; PI[0].T[2]=1;
PI[1].T[0]=1; PI[1].T[2]=1; PI[1].T[3]=2;
PIC[0].theAUT[0].recpt[0]=-1; PIC[0].theAUT[1].recpt[0]=0;
PIC[0].theAUT[2].recpt[0]=1; PIC[0].theAUT[3].recpt[0]=-1;
PIC[1].theAUT[0].recpt[0]=-1; PIC[1].theAUT[1].recpt[0]=-1;
PIC[1].theAUT[2].recpt[0]=3; PIC[1].theAUT[3].recpt[0]=0;
        Mask[0].T[0]=1; Mask[0].T[1]=1; Mask[1].T[0]=1;
        Mask[2].T[0]=1; Mask[2].T[1]=1; Mask[3].T[1]=1;
run Cont(0); run Cont(1); run Cont(2); run Cont(3);
        run A(); run B(); run C(); run D()
}
```

7 Conclusion

We have proposed a synthesis framework that produces an implementation for local choice HMSCs. This synthesis works with additional processes that tag messages and delay them to ensure correct ordering of message receptions. To keep the construction of CFSM simple, we have supposed FIFO semantics of communications, and we will hence suppose that the HMSCs that we implement do not contain message overtaking. However, the extension of our implementation to models that allow message overtaking should be easy. One fact worth mentioning again is that the controllers are purely asynchronous, which leaves a lot of freedom to choose a particular architecture. In a real implementation, one may suppose that a process and its controller are implemented on the same machine, but this is not mandatory. Controllers are designed to need as little information as possible to ensure that the processes they control are always executing a valid run of the specification : each process executes its task as defined in the projection of the specification, and controllers ensure coordination. In the future, we would like to study whether asynchronous controllers can in addition guarantee properties such as boundedness of some buffers, avoidance of a given configuration, etc.

Références

[1] B. Algayres, Y. Lejeune, F. Hugonment, and F. Hantz. The avalon project : a validation environment for sdl/msc descriptions. In *Proc. of SDL'93*, pages 221–235, 1993.

[2] R. Alur, G.J. Holzmann, and D. Peled. An analyser for mesage sequence charts. In *TACAS*, volume 1055 of *LNCS*, pages 35–48, 1996.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR*, pages 114–129, 1999.

[4] P. Baker, P. Bristow, C. Jervis, D.J King, and B. Mitchell. Automatic generation of conformance tests from message sequence charts. In *SAM*, pages 170–198, 2002.

[5] N. Baudru and R. Morin. Synthesis of safe message-passing systems. In *FSTTCS*, pages 277–289, 2007.

[6] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 259 – 274, April 1997.

[7] D. Brand and P. Zafiropoulo. On communicating finite state machines. Technical Report RZ1053, IBM Zurich Research Lab, 1981.

[8] Rational Software Corporation. Uml notation guide. Technical report, Rational Software Corporation, 1997. www.rational.com/uml.

[9] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level mscs : Model-checking and realizability. In *ICALP*, volume 2380 of *LNCS*, pages 657–668, 2002.

[10] L. Helouet. Sofat : Scenario formal analysis toolbox. Technical report, INRIA Rennes. `www.irisa.fr/distribcom/Prototypes/SOFAT/`.

[11] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from hmscs. In *5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 2000.

[12] G.J Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5) :279–295, 1997.

[13] ITU-T. Z.120 : Message sequence charts (msc). 1998.

[14] M. Mukund, K.N. Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from mscs. In *CONCUR*, pages 521–535, 2000.

[15] A. Muscholl, D. Peled, and Z. Su. Deciding properties for message sequence charts. In *FoSSaCS*, volume 1378 of *LNCS*, pages 226–242, 1998.

[16] B. Selic, G. Gullekson, and P.T. Ward. Real-time object-oriented modelling. *John Wiley & Sons, Inc*, 1994.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399