SOSYM manuscript No.
(will be inserted by the editor)

Distributed Implementation of Message Sequence Charts

Rouwaida Abdallah[1], **Loïc Hélouët**[2], **Claude Jard**[1]

[1] ENS Cachan – IRISA
Université Européenne de Bretagne
Campus de Ker-Lann, 35170 Bruz, France
e-mail: rouwaida.abdallah@bretagne.ens-cachan.fr

[2] INRIA – IRISA
Campus de Beaulieu, 35042 Rennes, France
e-mail: loic.helouet@inria.fr

[3] Université de Nantes - LINA
2 rue de la Houssinière, 44322 Nantes, France
e-mail: claude.jard@univ-nantes.fr

The date of receipt and acceptance will be inserted by the editor

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 scenarios, implementation, distributed system synthesis.

1 Introduction

Automatic program synthesis, that is deriving executable code from a high-level description of a problem (the requirements) is an old dream in computer science. The state of the art progresses through the invention of sophisticated compilation techniques which refine high-level specifications into lower level descriptions that are understandable by an execution platform.

This paper addresses the automatic synthesis problem in the difficult context of distributed applications running on networks of computers. The high-level formalism chosen for this study is Message Sequence Charts (MSCs

for short) [18]. MSCs have the advantage of specifying interactions patterns between components with a global perspective (such as an external observer) without worrying of how these interactions can be implemented locally by transmission and reception of messages in each component. The lower level for the synthesis is communicating finite automata [9] (CFSMs for short) exchanging messages asynchronously through FIFO channels. This well-known formalism is easily implementable on many distributed platforms built on top of standard communication protocols (TCP, REST, ...).

MSCs is a formal language based on composition of finite communication sequences. It is a prime example of scenario-based languages. It was standardized at the ITU, and has formed the basis for the formal definition of UML sequence diagrams [13]. The approach presented in this paper can certainly be adapted to other languages of this kind such as Use-Case Maps [20] or Live Sequence Charts [14], used for user requirements in distributed systems [4]. Basic MSCs (bMSCs for short) describe finite communication patterns involving a finite set of processes. Communications are supposed point to point and asynchronous. BMSCs are then composed using High-level MSCs (HMSCs for short), a finite automaton whose transitions are labeled by bMSCs. HMSCs are very expressive, and can model infinite state systems. Consequently, the usual verification techniques that apply to finite state machines (model checking of temporal logic formulae, intersecting two specifications to discover undesired behaviors,...) are in general undecidable [26, 10].

HMSCs are also difficult to implement. The main difficulty is to transform an HMSC specification with a *global* view (bMSCs contain complete messages, i.e. a message sending and its corresponding reception appear in the same diagram) of interactions into a set of *local* views (the communicating machines, which define disjoint sequences of messages sendings and receptions) that is consistent with the original specification. Consistency of an implementation w.r.t a specification can be defined in several ways. In this work, we impose that the synthesized model must exhibit the *same* behaviors as the original model. In the context of distributed systems modeling, it is natural to define these behaviors as a language of partial orders. Note that all the global coordination expressed by HMSCs can not always be translated to CFSMs. Consequently, some HMSC specifications may not be implementable as CFSMs. For instance, HMSCs allow for the definition of distributed choices, that is configurations in which distinct processes may choose to behave according to different scenarios. The HMSC semantics assumes a global coordination among processes, so all processes decide to execute the same scenario. However, when such distributed choice is implemented by local machines, each process may decide locally to execute a different scenario. When such unspecified situation occurs, the implementation is not consistent with the original HMSC: it exhibits more behaviors and even worse, the synthesized machines can deadlock. This should of course be avoided. HMSCs that do not contain distributed choices are called *local HMSCs*, and are considered as a reasonnable subclass to target

a distributed implementation. However, the synthesis solutions proposed so far do not apply to the whole class of local HMSCs.

This paper extends the state of the art by proposing an implementation mechanism for the whole subclass of local message sequence charts, that is HMSC specifications that do not require distributed consensus to be executed. Note that the class of local HMSCs seems a sensible specification model: Indeed, an HMSC which is not local obviously needs additional synchronization messages among participating processes, for instance to reach an agreement on which scenario to execute. One can see this need of additional messages as a transformation of the original non-local HMSC into a local one. Such transformation could be automated, but the solution is not unique. We will not study HMSC localization in this article. The proposed synthesis technique is to project an HMSC on each process participating to the specification. This technique is correct for a subclass of local HMSCs, namely the *reconstructible HMSCs*, but may produce programs with more behaviors than in the specification for local HMSCs that are not reconstructible [16]. When an HMSC is not reconstructible, we compose the projections with controllers, that intercept messages 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. One important aspect of this work is that processes and controllers are independent units of processing, which communicate asynchronously by means of delayed message passing and can be implemented on any distributed architecture. This provides a great genericity of the method.

This paper is organized as follows: Section 2 defines the formal models that will be used in the next sections. Section 3 characterizes the syntactic class of 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. Note that a large part of the material of sections 3 and 4 was already published in our former work [16]. We however include these results in order to obtain a self-contained article. Section 5 proposes a solution based on local control and message tagging to implement properly an HMSC. Section 6 presents a small use case, and section 7 compares our approach with existing techniques, and classifies it with respect to some criteria for scenario-based synthesis approaches. We then conclude and propose future research directions.

2 Definitions

In this section we first define our specification model, namely High-level Message Sequence Charts [18]. We then define our implementation model, communicating finite automata. We then list and justify the restrictions that were assumed for the specification model.

2.1 Basic MSCs, High-level MSCs

MSCs are composed of two specification layers: At the lowest level, basic MSCs define interactions among instances, and then these interactions are composed by means of High-level MSCs. A bMSC consists essentially of a finite set of processes (also called instances) denoted by I, that run in parallel and exchange messages in a one-to-one, asynchronous fashion. These instances represent different entities of a system such as processes, machines, etc. The life lines of instances are represented as vertical lines. They define sequences of *events*, ordered from top to bottom. An event can be a message sending or reception, or a local action. Horizontal arrows represent asynchronous messages from one instance to another. An example of bMSC involving three instances $\{Sender, Medium, Receiver\}$, three messages, and an internal action a is shown in Figure 1.

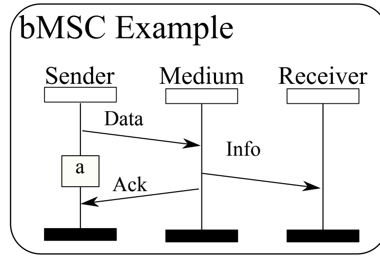

Fig. 1: An example of bMSC

Definition 1 (bMSCs) *A* bMSC *over a finite set of instances I is a tuple $M = (E, \leq, C, \phi, t, \mu)$ where:*

- *E is a finite set of events. The map $\phi : E \longrightarrow I$ localizes each event on an instance of I. 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 $\Sigma = \{p!q(m), p?q(m), a \mid p, q \in I, m, a \in C\}$. We have $t(e) = p!q(m)$ if $e \in E_p \cap S$ is a send event of message "m" from p to q, $t(e) = p?q(m)$ if $e \in E_p \cap R$ is a receive event of message "m" by p from q and $t(e) = a$ if $e \in E_p \cap L$ is a local action, named "a" located on p.*
- *$\mu : S \longrightarrow R$ is a bijection that matches send and receive events. If $\mu(e) = f$, then $t(e) = p!q(m)$ and $t(f) = q?p(m)$ for some $p, q \in I$ and $m \in C$.*

– *$\leq \subseteq E^2$ is a partial order relation (the "causal order"). It is required that events of the same instance are totally ordered: $\forall (e_1, e_2) \in E^2 \; \phi(e_1) = \phi(e_2) \Longrightarrow (e_1 \leq e_2) \vee (e_2 \leq e_1)$. For an instance p, let us call $\leq_p$ this total order. The causal ordering $\leq$ must also reflect the causality induced by the message exchanges, i.e. $\leq = (\bigcup_{p \in I} \leq_p \cup \mu)^*$*

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. 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. We will call $\phi(E)$ the set of *active instances* of M, and an instance will be called *minimal* if it carries a minimal event.

BMSCs allow for the compact definition of concurrent behaviors, but are limited to finite and deterministic interactions. To obtain infinite and non-deterministic specifications, we will use High-level MSCs, that compose sequentially bMSCs to obtain *languages of bMSCs*. Before defining HMSCs, we show how to assemble two bMSCs:

Definition 2 (Sequential composition) *Let $M_1 = (E_1, \leq_1, C_1, \phi_1, t_1, \mu_1)$ and $M_2 = (E_2, \leq_2, C_2, \phi_2, t_2, \mu_2)$ be two bMSCs, defined over disjoint sets of events. The* sequential composition *of M_1 and M_2 is denoted by $M_1 \circ M_2$. It consists in a concatenation of the two bMSCs instance by instance, and is obtained as follows. $M_1 \circ M_2 = (E, \leq, C, \phi, t, \mu)$, where:*

– *$E = E_1 \cup E_2, C = C_1 \cup C_2$*
– *$\forall e, e' \in E,\; e \leq e'$ iff $e \leq_1 e'$ or $e \leq_2 e'$ or $\exists (e_1, e_2) \in E_1 \times E_2 : \phi_1(e_1) = \phi_2(e_2) \wedge e \leq_1 e_1 \wedge e_2 \leq_2 e'$*
– *$\forall e \in E_1, \phi(e) = \phi_1(e),\; \mu(e) = \mu_1(e),\; t(e) = t_1(e)$*
– *$\forall e \in E_2, \phi(e) = \phi_2(e),\; \mu(e) = \mu_2(e),\; t(e) = t_2(e)$*

Note that the definition requires the concatenated bMSCs to be defined over disjoint sets of events. In the rest of the paper, we will use concatenation to assemble several occurrences of the same bMSC. Slightly abusing the definition, we will consider that concatenation $M_1 \circ M_2$ is always defined, and if $E_1 \cap E_2 \neq \emptyset$, we will consider that $M_1 \circ M_2$ is a bMSC obtained by composing M_1 with an isomorphic copy of M_2 defined over a set of events that is disjoint from E_1. In particular, this allows us to define, for a bMSC M, the bMSC $M \circ M$ which denotes a scenario with two consecutive occurrences of M. An intuitive and graphical interpretation for $M_1 \circ M_2$ is that the interactions in M_2 are appended to M_1 after M_1 (i.e. drawn below M_1). An example of sequential composition is shown in Figure 2: The bMSC $M_1 \circ M_2$ can simply be obtained by drawing M_2 below M_1, and extending the lifelines of instances. Note that sequential composition does not require both bMSCs to have the same set of instances.

HMSC diagrams are automata that compose bMSCs or other HMSCs. This allows for the definition of iterations and choices. We will suppose without loss of generality that our HMSCs comprise only one hierarchical level, i.e. they are automata whose transitions are labeled by bMSCs.

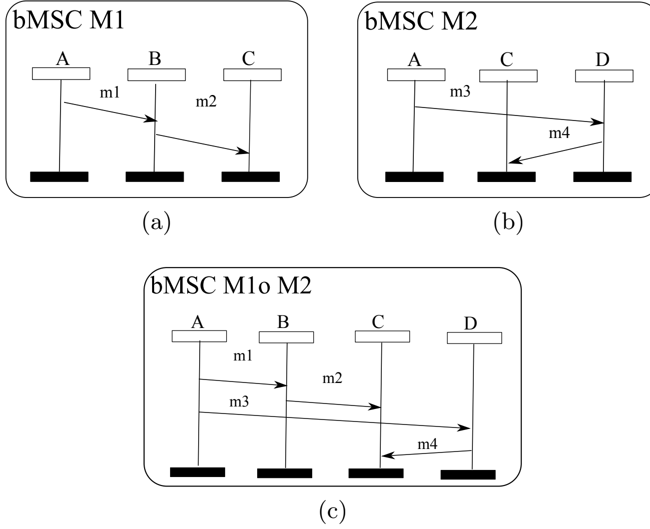

Fig. 2: Two bMSCs M_1 and M_2 and their concatenation $M_1 \circ M_2$

Definition 3 (HMSCs) *An* 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.*

HMSCs contain a unique *initial node* n_0, that has no incoming transition (i.e, there is no transition of the form (n, M, n_0) in $\rightarrow$), but also *sink nodes*, i.e. nodes that have no successor, and *choice nodes*, i.e. nodes that have several successors. Figure 3 shows an example of HMSC with two nodes n_0, n_1, where n_0 is the initial node (and also a choice node), and n_1 is a sink node. A node is represented by a circle. Initial nodes have a downward pointing triangle connected to them, and sink nodes are connected to an upward pointing triangle. In this example, we have $\mathcal{M} = \{M_1, M_2\}$ where M_1, M_2 are the bMSCs defined in Figure 2. The transition relation contains two transitions, namely (n_0, M_1, n_0) and (n_0, M_2, n_1). In the example of Figure 3, the behavior M_1 can be repeated an arbitrary number of times, and then be followed by the behavior described in M_2.

For convenience, we will consider that all nodes, except possibly the initial node and sink nodes are choice nodes, i.e. have several successors by the transition relation. This results in no loss of generality, as an HMSC can be always transformed in such a canonical form by concatenating bMSCs appearing in a path. A transition from a (choice) node will be frequently called *a branch* of this choice. We also require HMSCs to be deterministic,

that is if $(n, M_1, n_1) \in\longrightarrow \wedge (n, M_2, n_2) \in\longrightarrow$, then $M_1 \neq M_2$. This can be ensured by the standard determinization procedure of finite automata.

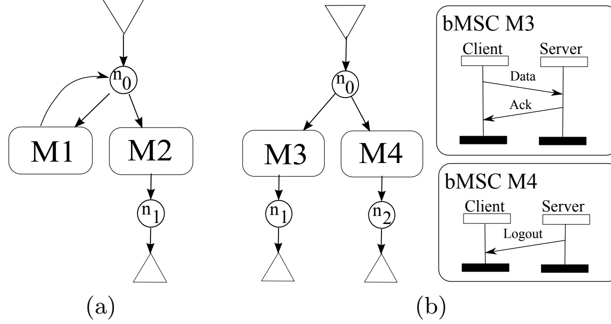

Fig. 3: An example of local HMSC a) and a non-local HMSC b)

2.2 Semantics of HMSCs

In the next sections, executions of bMSCs will be represented as partially ordered multisets of events (*pomsets*). These pomsets are not necessarily bMSCs, as we will consider incomplete executions in which some messages have been sent and not yet received. This notion of incomplete execution is captured by the definition of *pieces* and *prefixes*.

Definition 4 (prefix, suffix, piece of bMSCs) *Let $M = (E, \leq, C, \phi, t, \mu)$ be a bMSC. A* prefix *of M is a tuple $(E', \leq', C', \phi', t', \mu')$ such that E' is a subset of E closed by causal precedence (i.e. $e \in E' \wedge f \leq e \Longrightarrow f \in E'$) and $\leq', C', \phi', t', \mu'$ are restrictions of $\leq, C, \phi, t, \mu$ to E'. A* suffix *of M is a tuple $(E', \leq', C', \phi', t', \mu')$ such that E' is closed by causal succession (i.e. $e \in E' \wedge\ e \leq f \Longrightarrow f \in E'$) and $\leq', C', \phi', t', \mu'$ are restrictions of $\leq, C, \phi, t, \mu$ 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 their message mappings m are not necessarily bijections 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. We will denote by O_ϵ the empty prefix, i.e. the prefix that contains no event. For a particular type of action a, we will denote by O_a a piece containing a single event of type a. The examples of Figure 4 shows a bMSC M involving three processes P, Q, R, a prefix Pr, a suffix S, and a piece Pc. Observe that Pc is obtained by erasing Pr and S from M. Note also that Pr, S and Pc contain incomplete messages.

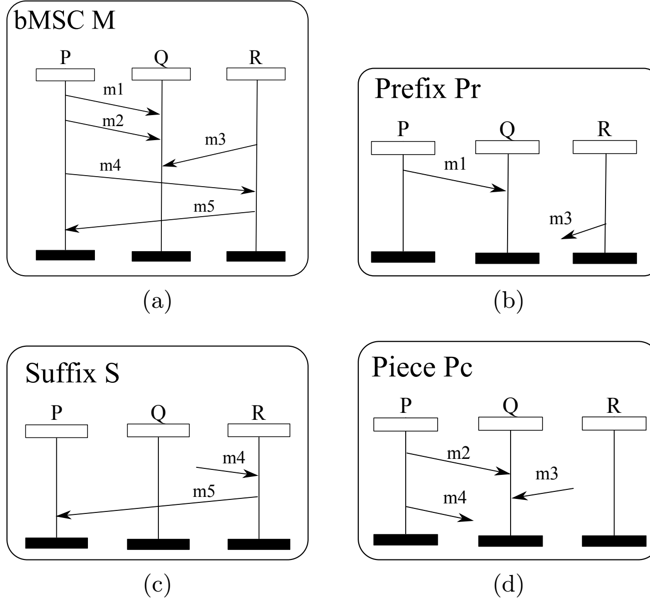

Fig. 4: A bMSC (a), a prefix (b), a suffix (c) and a piece (d)

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 bMSC. Then, the concatenation of O_1 and O_2 is denoted by $O_1 \circ O_2 = (E, \leq, C, \phi, t, \mu)$, where $E, \leq, C, \phi$, and t are defined as in definition 2 and μ is a function that associates the nth sending event from p to q to the nth 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 nth sending and reception do not match, that is one event is of the form $p!q(m)$ and the other one $q?p(m')$ with $m \neq m'$. In particular, we will denote by $O \circ \{e\}$ the prefix obtained by concatenation of a single event e to a prefix O.

Definition 5 (HMSC behavior) *Let $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ be an HMSC. A* path *of H is a sequence $\rho = (n_0, M_0, n_1)(n_1, M_1, n_2) \dots (n_k, M_k, n_{k+1})$ of transitions. We will say that a path is* acyclic *if and only if it does not contain the same transition twice. We define as $Paths(H)$ the set of paths of H starting from the initial node. A path $\rho = (n_0, M_0, n_1) \dots (n_k, M_k, n_{k+1})$ in $Paths(H)$ defines a sequence $M_0.M_1 \dots M_k \in \mathcal{M}^*$ of bMSCs. We will denote by O_ρ the bMSC associated to ρ and define it as $O_\rho = M_0 \circ M_1 \circ \dots \circ M_k$. The* language *of H is the set of behaviors $\mathcal{L}(H) = \bigcup_{\rho \in Paths(H)} Pref(O_\rho)$.*

To simplify notation, we will write $\rho = n_0 \xrightarrow{M_0} n_1 \xrightarrow{M_1} n_2 \dots \xrightarrow{M_k} n_{k+1}$ to denote a path $\rho = (n_0, M_0, n_1)(n_1, M_1, n_2) \dots (n_k, M_k, n_{k+1})$. Note that

our definition of the language of an HMSC H includes all prefixes of bMSCs generated by H. Note also that the language of an HMSC is not a regular language in general. In the example of Figure 3, for instance, behaviors in which machine A asynchronously sends an arbitrary number of messages without waiting for their reception are contained in the language of the specification. Clearly, such behaviors can not be represented as a regular language. A *correct implementation* of an HMSC H is a distributed system reproducing **exactly** (and nothing more) $\mathcal{L}(H)$.

2.3 Communicating Finite State Machines

In this section, we introduce our implementation model, namely Communicating Finite State Machines (CFSM) [9]. A CFSM $\mathcal{A}$ is a network of finite state machines that communicate over unbounded, non-lossy, error-free and FIFO communication channels. Before formally introducing CFSM, let us justify the choice of this model as implementation formalism. First of all, HMSCs describe the behavior of independent agents, which continuously run sequences of communication events. Hence, if we want to respect the independence of agents in the architecture depicted in an HMSC, we need a target model that allows for the definition of parallel components. Models such as Petri nets or networks of automata communicating via shared actions fulfill these requirements. However, it is clearly stated in the Z.120 standard [18] that bMSCs and HMSCs depict the behavior of agents that communicate asynchronously, which rules out communications using shared actions. One can also notice that the projection of an HMSC on a single instance gives a regular language. Last, one shall notice that HMSCs semantics can enforce messages between a pair of processes to respect FIFO ordering, which can not be enforced by Petri nets. In fact, it has been shown that synthesis of Petri nets from HMSCs usually produces an overapproximation of the initial HMSC language [10]. All these considerations call for the use of CFSMs as target architecture.

We will write $\mathcal{A} = \underset{i \in I}{\|} A_i$ 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. More formally, we can define a communicating automaton as follows:

Definition 6 *A* communicating automaton *associated to an instance p is a tuple $A_p = (Q_p, \delta_p, \Sigma_p, q_{0,p})$ where Q_p is a set of states, $q_{0,p}$ is the initial state, Σ_p is an alphabet with all letters of the form $p!q(m)$ $p?q(m)$ or a, symbolizing message sending to a process q, reception from a process q, an atomic action a executed by process p, or a silent move ϵ. The transition relation $\delta_p \subseteq Q_p \times \Sigma_p \times Q_p$ is composed of triples (q, σ, q') indicating that the machine moves from state q to state q' when executing action σ. A* CFSM *$\mathcal{A} = \underset{i \in I}{\|} A_i$ is a composition of communicating automata.*

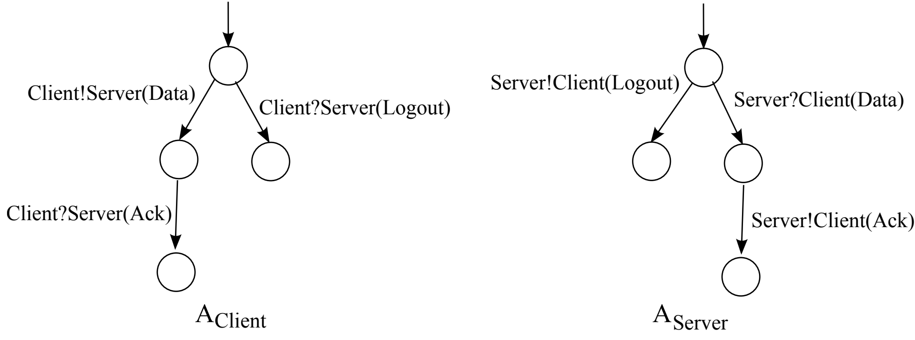

Fig. 5: Two communicating machines

Figure 5 describes a CFSM composed of two finite state machines A_{Client} and A_{Server}. The initial states of these two machines are denoted by a dark incoming arrow. Each run of a 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 sendings and receptions). Then, the language $\mathcal{L}(\mathcal{A})$ of a set of communicating machines is the set of all prefixes associated to runs of $\mathcal{A}$.

The semantics of CFSM is usually defined as sequences of events. Each event occurs on a single process, and changes the configuration of the CFSM. A *configuration* of a network of automata $\mathcal{A} = \underset{i \in I}{\|} A_i$ is a pair $C = (L, W)$ where L is a sequence of states $q_1 \dots q_I$ depicting the local state of each communicating machine, and $W = \{w_{11}, \dots w_{1|I|}, w_{21}, \dots w_{2|I|}, \dots w_{|I||I|}\}$ is a set of $|I|^2$ words depicting the contents of message buffers. Each w_{ij} is a sequence of message names, and depicts the contents of the queue from A_i to A_j. Then, the behavior of $\mathcal{A}$ is defined as follows:

- all machines start from their initial states with all communication buffers empty, that is the initial configuration is $C_0 = (L_0 = q_{0,1}. \dots .q_{0,|I|}, W_0 = \{\epsilon, \dots \epsilon\})$.
- From a configuration C, a machine A_p can send a message m to a machine A_q if A_p is in local state q_p, there exists a transition $(q_p, p!q(m), q'_p)$ in A_p. Executing this action $p!q(m)$ simply appends m to the buffer $w_{p,q}$ from p to q and changes A_p's local state to q'_p in the configuration. Hence, if $C = (L, W)$ with $L = q_0 \dots q_p \dots q_{|I|}$ and $W = \{w_{11}, \dots w_{p,q} \dots w_{|I||I|}\}$, executing $p!q(m)$ results in a configuration $C' = (L', W')$ with $L' = q_0 \dots q'_p \dots q_{|I|}$ and $W' = \{w_{11}, \dots w_{p,q}.m \dots w_{|I||I|}\}$ Local actions of communicating automata change the local state of a machine and leave the buffer contents unchanged.
- From a configuration C, A_p can receive a message m from process q, if A_p is in local state q_p, there exists a transition $(q_p, p?q(m), q'_p)$ in A_p,

and the first letter of $w_{q,p}$ is m (which means that m is the first message that has to be received in the queue from q to p). Executing this action $p?q(m)$ simply removes m from the buffer $w_{p,q}$ from p to q and changes A_p's local state to q'_p in the configuration. Hence, if $C = (L, W)$ with $L = q_0 \dots q_p \dots q_{|I|}$ and $W = \{w_1 1, \dots w_{p,q} = m.w \dots w_{|I||I|}\}$, executing $p?q(m)$ results in a configuration $C' = (L', W')$ with $L' = q_0 \dots q'_p \dots q_{|I|}$ and $W' = \{w_{11}, \dots w_{p,q} = w \dots w_{|I||I|}\}$.

This way, CFSMs define sequences of actions $\sigma_1 \dots . \sigma_k$ that can be executed by their local components from their initial states. Each action moves the communicating machines from one configuration to another. However, CFSM are concurrent models, and their executions can be represented in a non-interleaved way by bMSC prefixes.

Definition 7 *Let $\mathcal{A} = \underset{i \in I}{\|} A_i$ be a CFSM. The language of $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$ and is the set of prefixes defined inductively as follows :*

- *the prefix associated to an empty sequence of actions is the empty prefix O_ϵ,*
- *the prefix associated to a sequence of actions $\sigma_1 \dots . \sigma_k . \sigma_{k+1}$ of $\mathcal{A}$ is the prefix $O \circ \{e\}$ where e is an event labeled by σ_{k+1} and O is the prefix associated to $\sigma_1 \dots . \sigma_k$.*

2.4 Restrictions

We have assumed some restrictions on the scenarios that we implement. Some of them are introduced for the sake of readability, and some of them are essential to ensure a solution to the synthesis problem. Standard notation of bMSCs allow for the definition of a zone on an instance axis called *co-region*. Events appearing in a co-region can be executed in any order. We do not consider co-regions, but they can be simulated by adding to an HMSC a finite number of alternatives enumerating all possible interleavings of events. We also consider that HMSCs are deterministic, and that two bMSCs labeling distinct transitions of a local HMSC start with distinct messages. We use this assumption to differentiate branches at runtime. We could achieve a similar result by introducing additional tags during synthesis. However, this mild restriction simplifies the notations and proofs.

BMSCs also allow behaviors with *message overtaking*, i.e. in which some messages mandatorily cross other messages from the same bMSC. In this paper, we consider only FIFO architectures as a target for synthesis. This is hence a natural restriction to consider that all bMSCs are FIFO, that is for two sending events e, e' such that $p = \phi(e) = \phi(e')$, $q = \phi(\mu(e)) = \phi(\mu(e'))$ we always have $e \leq_p e' \iff \mu(e) \leq_q \mu(e')$. Note that our synthesis technique could be easily adapted to allow overtaking in bMSCs. This requires a slight modification of the communication architecture, to allow a bounded lookahead at the contents of communication buffers, and consumption of

messages appearing at a fixed position in a FIFO buffer rather that in first position. Such semantics exists for instance in extended automata models such as SDL, and a synthesis technique to generate SDL code from HMSCs in which bMSCs contain message crossings was proposed in [1].

We restrict to HMSCs without parallel frames for deeper reasons. When parallel frames are used, the behavior of an agent may not be a regular language, i.e. it may not be expressible as a finite state machine. The implementation technique proposed in this paper uses vectorial clocks that may grow unboundedly, but the systems generated always comport a finite number of control states. Furthermore, the use of parallel frames may add a new source of unexpected behaviors, as one agent may have to react differently when a pair of actions a, b are executed concurrently or in sequence, and such non-determinism may lead to the execution of unspecified behaviors. Hence, we doubt that a simple machine model can handle at the same time unbounded parallelism in agents and asynchronous communications, to implement the extremely complex (and very often ambiguous) behaviors allowed with parallel frames. Such extension to CFSMs goes beyond the scope of this paper.

3 Local HMSCs

Consider a choice node in an HMSC, that is a node n with at least two outgoing transitions (n, M_1, n_1) and (n, M_2, n_2). Executing an event in M_1 (resp. M_2) can be seen as taking the decision to execute the whole behavior contained in M_1 (resp. M_2). Once the decision to perform M_1 or M_2 is taken, all the other instances have to conform to this decision to remain consistent with the HMSC specification. Hence, every bMSC M_i labeling a transition leaving a choice node defines a set of *deciding instances* $\phi(Min(M_i))$, which is the set of instances that carry the minimal events of M_i, and hence can take the decision to perform bMSC M_i. Obviously, the minimal events in each M_i cannot be message receptions.

We can now state the main difficulty when moving from HMSCs to local machines. In an HMSC, the possible executions are built by concatenating bMSCs one after another. Hence in an execution of an HMSC, all processes conform to a single sequence of bMSCs collected along a path. In a CFSM setting, when two processes have to take a decision to perform scenario M_1 or M_2, they can of course take concurrently the same decision, but conversely, one instance can decide to perform scenario M_1 while the other instance decides to perform M_2. Consider for instance the HMSC of Figure 3-b. The instance Client can decide to send *Data* and wait for an acknowledgment while the instance Server decides to send *Logout*. Such situation can lead to a deadlock of the system. Even worse, this scenario was not specified in the original description. Such unspecified scenarios are frequently called “implied scenarios”, and were originally studied in [30]. The main intuition behind this notion of implied scenario is that even though a

scenario was not part of the original specification H, as a distributed implementation of H can execute it, then it should be considered as part of the specification, and explicitly appended to the original model [31]. This approach may work for simple cases, but not for all kinds of HMSCs. First of all, an HMSC may exhibit an infinite number of implied scenarios. Furthermore, it is undecidable if an implied scenario is a prefix of some run that already exists in the original specification (this problem can be brought back to a language inclusion problem for HMSCs, which was shown to be undecidable [26,10]). So, one can not decide if a specification already includes an implied scenario that was discovered for a particular choice node. Furthermore, every implied behavior appended to an HMSC may produce new implied scenarios and the growth of a specification due to the integration of these new behaviors may never stop. A safer design choice is to consider that situations leading to non-local choices and hence to implied scenarios have to be avoided. For this, we define local HMSCs.

When the outgoing transitions of a choice node are labeled by bMSCs with distinct deciding instances, then, without additional synchronization the synthesized machines might decide to perform distinct scenarios. This situation is called *non-local choice*, and should be avoided in a specification. We consider that specifications containing non-local choices are not refined enough to be implemented.

Definition 8 (Local choice node) *Let* $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ *be an HMSC, and let* $c \in N$ *be a choice node of* H*. Choice* c *is* local *if and only if for every pair of (not 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 *HMSC if all its choices are local.*

We will also say that an HMSC is *non-local* if one of its choices is not local. Intuitively, the locality property described in [6] guarantees that every choice is controlled by a unique instance. We will show however that ensuring locality of choices is not sufficient to guarantee a correct synthesis.

Proposition 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 exists a node c_1 and two paths ρ_1, ρ_1' such that at least one (say ρ_1) of them is not acyclic, and ends with transitions that appear several times along this path. 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 for all bMSCs M, $\phi(min(M \circ M)) = \phi(min(M))$.

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 1 (Complexity of local choices) *Deciding if an HMSC is local 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 bMSCs 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 bMSCs 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 $Min(M) = E \setminus \{f \mid \exists e, e \leq_p f \vee f = \mu(e)\}$. □

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 paths start is not local. Note that the set of minimal instances on a path ρ (or the whole bMSC O_ρ labeling this path) needs not be recomputed everytime a path is extended, 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 that is investigated.

Algorithm 1 was originally proposed in [16]. 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. The correctness of the algorithm is guaranteed by theorem 1, and as we consider a finite set of maximal acyclic paths, termination is guaranteed.

4 The Synthesis Problem

The objective of the synthesis algorithm from an HMSC H is to obtain a CFSM $\mathcal{A}$ that behaves exactly as H. An obvious solution is to project the original HMSC on each instance, that is if H is defined over a set of instances I, we want to build a CFSM $\mathcal{A} = \parallel_{i \in I} A_i$ such that $\mathcal{L}(H) = \mathcal{L}(\mathcal{A})$.

Algorithm 1 LocalChoice(H)

for c node of H **do**
  P $= \{(t, I, J) \mid t = (c, M, n) \wedge I = \phi(min(M)) \wedge J = \phi(M)\}$
  /*P contains acyclic paths*/
  MAP $= \emptyset$ ***/*Maximal acyclic paths*/***

  while $P \neq \emptyset$ **do**
    $$\text{MAP} = \text{MAP} \cup \left\{ \begin{array}{c} (w.t, I) \mid \exists (w, I, J) \in P, \exists t = (n_k, M, n) \in w, \\ w = t_1...t_k \wedge t_k = (n_{k-1}, M_k, n_k) \end{array} \right\}$$
    $$\text{P} = \left\{ \begin{array}{l} (w.t, I', J') \mid \exists (w, I, J) \in P, \exists t = (n_k, M, n) \in \longrightarrow, \\ \qquad w = t_1...t_k \wedge t_k = (n_{k-1}, M_k, n_k), \\ \qquad \wedge t \notin w \wedge J' = J \cup \phi(M) \wedge I' = I \cup (\phi(min(M)) - J) \end{array} \right\}$$
  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

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. This operation preserves the structure of the HMSC automaton: Starting from an automaton labeled by bMSCs, we obtain an automaton labeled by (possibly empty) sequences of events located on the considered instance. 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 (see for instance chapter 2.4 of [17]).

Definition 9 (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* $E_{M\,i}$*. The set* $E_{M\,i}$ *is totally ordered by* $\leq_i$*. We denote its elements by* $e_1, \cdots, e_{|E_{M\,i}|}$*. The finite state automaton* A_i*, result of the projection of* H *onto the instance* i *is* $A_i = (Q_i, \rightarrow_i, E_i \cup \{\varepsilon\}, n_0)$*. We encode states of* A_i *as tuples* $(n, M, n', k) \in N \times \mathcal{M} \times N \times \mathbb{N}$*, where the first three components designate an HMSC transition labeled by a bMSC* M *defined over a set of events* E_M*, and the last component* k *is an index ranging from* 1 *to* $|E_{M_i}|$ *indicating the progress of instance* i *during* M*, or simply as a reference to an HMSC node* n *(designating a configuration in which* A_i *has not yet started the execution of a bMSC from* n*). We then have* $Q_i = \{n\} \cup \{(n, M, n', k) \mid (n, M, n') \in \longrightarrow \wedge\ k < |E_{M_i}|\}$,

and $E_i = \bigcup_{M \in \mathcal{M}} E_{M\,i}$. *We can then define the transition relation* $\longrightarrow_i$ *as*
$\longrightarrow_i = \{(n, \epsilon, n') \mid \exists (n, M, n') \in \longrightarrow \wedge |E_{M_i}| = 0\}$
$\cup \{(n, t(e_1), n') \mid \exists (n, M, n') \in \longrightarrow \wedge |E_{M_i}| = 1\}$
$\cup \{(n, t(e_1), (n, M, n', 1)) \mid (n, M, n') \in \longrightarrow \wedge |E_{M_i}| \geq 2\}$
$\cup \{((n, M, n', k-1), t(e_k), (n, M, n', k)) \mid (n, M, n') \in \longrightarrow \wedge 2 \leq k < |E_{M_i}|\}$
$\cup \{((n, M, n', k-1), t(e_k), n') \mid (n, M, n') \in \longrightarrow \wedge k = |E_{M_i}|\}$

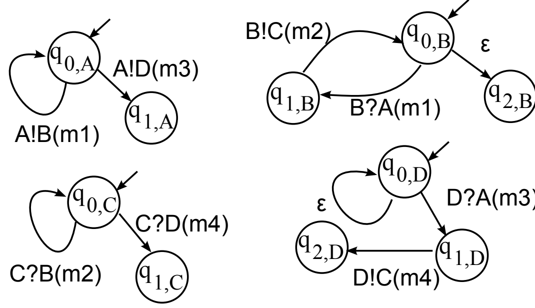

Fig. 6: The instance automata projected from the HMSC of Fig. 3-a).

The synthesis by projection from the HMSC of Figure 3-a) produces the CFSM of Figure 6. Note that as instance D is not active in bMSC $M1$, there is an ϵ-transition in the automaton associated to D. The synthesis from the HMSC of Figure 3-b) produces the CFSM of Figure 5. In this model, the CFSM can behave as specified in scenarios M_1 and M_2. However, A_{client} can also decide to send a *Data* message while A_{Server} sends a logout message. This situation was not specified in the HMSC of Figure 3-b), so the CFSM of Figure 5 cannot be considered as a correct implementation. In general, the projection of an HMSC on its instances can define more behaviors than the original specification, but can also deadlock. Hence, synthesis by projection on instance is not correct for any kind of HMSC. It was proved in [16] that the synthesized language contains all runs of the HMSC specification.

Theorem 2 ([16]) *Let H be an HMSC and let $\mathcal{A}$ be the CFSM obtained by projection of H on its instances. Then $\mathcal{L}(H) \subseteq \mathcal{L}(\mathcal{A})$.*

In the rest of the paper, we will only consider local HMSCs. However, this is not sufficient to ensure correctness of synthesis. Let us consider the projection of H in Figure 3 on all its instances given in Figure 6. A correct behavior of H is shown in Figure 7-a), while a possible but incorrect behavior of the synthesized automata is shown in Figure 7-b). We can see that message m_4 sent by machine D can arrive at machine C while m_2 sent by machine B is still in transit. According to the HMSC semantics, machine C should delay the consumption of m_4 to receive message m_2 first. However,

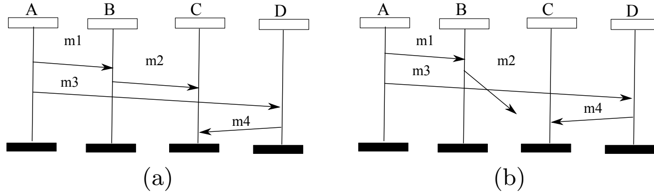

Fig. 7: *a*) A correct behavior of the HMSC of Fig. 3-a), and *b*) a possible distortion due to the loss of information on projected instances.

C does not have enough information to decide to delay the consumption of m_4, and hence exhibits an unspecified behavior.

This example proves that in general, even for local HMSCs, the synthesis by projection is not correct. Problems arise when an instance does not have enough information on the sequences of choices that have occurred in the causal past of a message reception event. In some sense, the projection of an HMSC on local components breaks the global coordination between deciding instances and the other instances in the system.

Definition 10 *Let H be a local HMSC and c be a choice node of H. Let ρ be a cyclic path starting from c, and ρ' be any acyclic path starting from c. Let H_c be the HMSC with two nodes c, c', two transitions (c, O_ρ, c) and (c, O'_ρ, c'). Let $\mathcal{A}_c$ be the CFSM obtained by projection from H_c. We will say that c, ρ, ρ' is a* sequence-loss witness *iff $\mathcal{L}(H_c) \neq \mathcal{L}(\mathcal{A}_c)$.*

We will say that an HMSC is *reconstructible* if and only if it is local and has no sequence-loss witnesses. The class of reconstructible HMSCs was proposed in [16]. This paper also shows that it is sufficient to consider simple cycles leaving a choice to detect sequence-loss witnesses, which allows for the definition of a terminating algorithm. Furthermore, one does not have to simulate all runs of communicating automata in $\mathcal{A}_c$ to detect that $\mathcal{L}(H_c) \neq \mathcal{L}(\mathcal{A}_c)$. Indeed, sequence losses can be detected by checking if the sequential ordering of events along a non-deciding instance in prefix $O_\rho \circ O_{\rho'}$ can be lost during projection. To avoid technical details, we will not show in this paper how the sequence losses can be found from $O_\rho \circ O_{\rho'}$, but rather illustrate the approach on an example. We refer interested readers to [16] for formal details. Let us consider the example of Figure 3-a), with a single choice node n_0, and the path $(n_0, M_1, n_0).(n_0, M_2, n_1)$. According to the semantics of HMSCs, reception of messages m_2 and m_4 on instance C should occur in this order in a correct implementation of the example. Now let us consider the automata obtained by projection of H on instances, as in Figure 6. After executing $A!B(m1).B?A(m1).B!C(m2).A!D(m3).D?A(m3).D!C(m4)$, the CFSM is in configuration $(L = q_{1,A}.q_{0,B}.q_{0,C}.q_{2,D}, W = \{\epsilon, \dots w_{BC} = m2, w_{DC} = m4, \dots \epsilon\})$. From this configuration, the automaton corresponding to instance C can receive $m2$, which is the expected behavior, or conversely

receive $m4$ which is wrong according to the choices that were performed by instance A. Hence $n_0, (n_0, M_1, n_0), (n_0, M_2, n_1)$ is a sequence loss witness. This can be easily seen from $M_1 \circ M_2$: If one removes the ordering between the reception of $m2$ and the reception of $m4$, there is no way to infer this ordering from remaining causalities. One important fact is that synthesis by projection is correct for the subclass of reconstructible HMSCs.

Theorem 3 ([16]) *Let H be a reconstructible HMSC, and $\mathcal{A}$ be the CFSM obtained from H by projection. Then, $\mathcal{L}(H) = \mathcal{L}(\mathcal{A})$.*

As for local HMSCs, one can easily show that detecting if an HMSC is reconstructible is a $co-NP$ problem. According to theorem 3, the communicating automata synthesized from reconstructible HMSCs are correct implementations. However, we show in the next section, that all local HMSCs can be implemented with the help of additional controllers. This allows for the following synthesis approach: first check if an HMSC is reconstructible. If the answer is yes, then synthesize the CFSM by simple projection as proposed in section 4. If the answer is no, then synthesize the CFSM with their controllers, as proposed in section 5.

5 Implementing HMSCs with message controllers

The class of reconstructible HMSCs shown in section 4 is contained in the class of local HMSCs. This subclass is quite restrictive (for instance, the HMSC of Figure 3-a) is not reconstructible, and hence can not be implemented by a simple projection). Note also that the difference between the languages of an HMSC and of the synthesized machines comes from the fact that some communicating automata consume a wrong message instead of waiting for the arrival of the message specified by the HMSC. In this section, we address the synthesis problem in a different setting, that is we add a local controller to each communicating machine that can tag messages and delay their delivery. As synthesis fails because of reception of messages in the wrong order, each controller will receive messages destinated to the machine it controls, and decide whether it should deliver it or delay its delivery. This decision is taken depending on additional information carried by messages, namely a vector clock. Vector clocks is a well known mechanism, and helps keeping track of global progress in distributed systems.

This new mechanism allows 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. In every automaton $K(A_i)$ we replace each transition of the form $\big((n_1, M_1, k, n_2), p!q(m), (n_3, M_2, k', n_4)\big)$ (respectively $\big((n_1, M_1, k, n_2), p?q(m), (n_3, M_2, k', n_4)\big)$) in A_i, by a transition of the form $\big((n_1, M_1, k, n_2), p!C_p(q, m, b), (n_3, M_2, k', n_4)\big)$ (respectively $\big((n_1, M_1, k, n_2), p?C_p(q, m, b), (n_3, M_2, k', n_4)\big)$), where b indicates the branch to which the sending or the reception belongs. A controller C_i can receive messages of the form (q, m, b) from his controlled process $K(A_i)$. In such cases, it tags them with a clock (the contents of this clock is defined later in this section), and sends them to controller C_q. Similarly, each controller C_i 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 (i.e. left intact in buffer). Automata and their controllers communicate via FIFO channels, which defines a total ordering on message receptions or sendings. Controllers also exchange their tagged messages via FIFO buffering. 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, and a set of controllers $\{C_i\}_{1\leq i\leq n}$. Each communicating automaton $K(A_i)$ is connected via a bidirectional FIFO channel to its associated controller C_i. The controllers are themselves interconnected via a complete graph of bidirectional FIFO channels. We will refer to these connections among communicating automata as *ports*. A machine $K(A_i)$ communicates with its controller via a port P_i, and for all $i \neq j$, port $P_{i,j}$ of controller C_i is connected to the port $P_{j,i}$ of controller C_j. This architecture is illustrated in Figure 8 for three processes i, j, k. 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.

5.2 Tagging mechanism

Vector clocks are a standard mechanism to record faithfully executions of distributed systems (see for instance [11,24]), or to enforce some ordering on communication events [27]. Usually, vector clocks count events that have occurred on each process. In the architecture that we defined, each controller maintains a vector clock that counts the number of occurrences of each branch of an execution it is aware of.

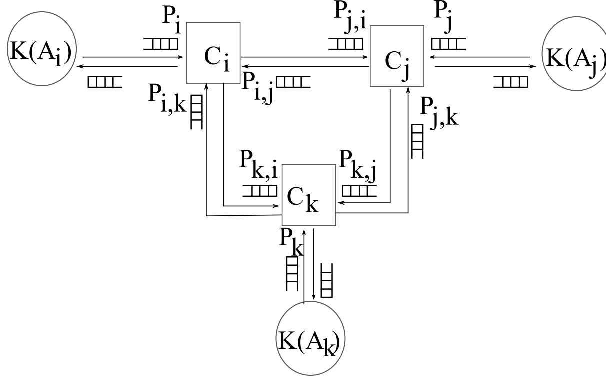

Fig. 8: The distributed controlled architecture.

To allow for faithful recording of branches chosen along an execution we have to set up a total ordering on branches of HMSCs. Let H be an HMSC. We will denote by $\mathcal{B}_H$ the branches of H, and fix an arbitrary total ordering $\lhd$ on $\mathcal{B}_H$. We use this arbitrary order on branches to index integer vectors that remember the number of occurrences of branches that have occurred during an execution of an HMSC. Let us consider the example of Figure 3, that contains two branches $b_1 = (n_0, M_1, n_0)$ and $b_2 = (n_0, M_2, n_1)$. We can fix $b_1 \lhd b_2$, and associate to every execution a vector τ of two integers, where $\tau[b_i], i \in 1, 2$ represents the number of occurrences of branch b_i in the execution.

Definition 11 (Choice clocks) *A* choice clock *of an HMSC H is a vector of* $\mathbb{N}^{\mathcal{B}_H}$. *Let* $\rho = n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \dots \xrightarrow{M_k} n_k$ *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* $M_1 \circ \dots \circ M_i$.

Intuitively, choice clocks count the number of occurrences of each choice in a path of H. In the rest of this section, we will show that communicating automata and their controllers can maintain *locally* a choice clock along the prefix that they are executing, and that choice clocks carry all the needed information to forbid the execution of prefixes that are not in $\mathcal{L}(H)$.

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'. We will say that vectors V_1 and V_2 are equal, denoted $V_1 = V_2$, if $V_1[b] = V_2[b]$ for every entry b. We will say that V_2 is greater than V_1, denoted $V_1 \prec V_2$, 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 $\rho = n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \dots \xrightarrow{M_k} n_k$, 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 bMSC M_i. From these facts, the following proposition is straightforward:

Proposition 2 *Let H be a local HMSC, ρ be a path of H, and τ be the choice clock labeling of O_ρ. Then, $(\tau(E_{O_\rho}), \prec)$ is a totally ordered set.*

This proposition is important: maintaining locally a consistent tagging of messages allows a controller that has two tagged messages available in two of its buffers to decide which one should be delivered first.

Definition 12 (Concerned instances) *Let $b = (c, M, n)$ be a branch of an HMSC H. We will say that instance $p \in I$ is* concerned *by branch b if and only if there exists an event of M on p ($E_{Mp} \neq \emptyset$). Let $K \in \mathbb{N}^{\mathcal{B}_H}$ be a choice clock, and let $p \in I$ 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 $[K]_p$.*

In the example in Figure 3, the choice clock is a integer vector indexed by b_1, b_2, where $b_1 = (n_0, M_1, n_0)$ and $b_2 = (n_0, M_2, n_1)$. Considering M_1 and M_2 as defined in Figure 2, instances A, C are concerned by both branches (they are active in M_1 and M_2), but instance B is concerned only by b_1 and instance D is concerned only by b_2.

For a given instance $i \in I$, 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. Messages exchanged between the automata and the controllers are triples (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 *locally known choices* vector. It is initialized to the null vector, and updated upon consumption of incoming messages.
- *numEvt*, 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. *numEvt* 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 $i \in I$, and by $\pi_{i,?}(M)$ the restriction of this sequence to receptions. For a sequence of events w, we will

denote by $tail(w)$ the sequence of events obtained by removing the first event from w, that is if $w = a.v$, then $tail(w) = v$. The generic algorithm for a controller C_i is composed of two rules, which are always active (see Algorithm 2). 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 *numEvt* 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** $P_{i,j}, j \neq i$, and aims at controlling the order of the different receptions of messages arriving in the buffers between each controller $C_j, j \in I \setminus \{i\}$ and controller C_i. This is the main objective of the controller. Note that these messages arrive in a distinct buffer for each neighbor 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 the 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 $P_{i,j}$ 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.

The algorithm 2 executed by every controller is presented next page.

Algorithm 2 Controller C_i

RULE 1: when (j, m, b) available on port P_i
/* There is a message from $K(A_i)$ in the buffer from $K(A_i)$ to C_i */
consume (j, m, b)
if $numEvt = 0$ **then**
  $\tau_i[b]{+}{+}$
  $numEvt := |\Pi_i(M_b)| - 1$
  $Rec = \Pi_{i,?}(M_b)$
  send (m, τ_i) to C_j via port $P_{i,j}$
else
  $numEvt$ - -
  send (m, τ_i) to C_j via port $P_{i,j}$
end if

RULE 2: when there exists a port $P_{i,j}$ with (m, τ) available on port $P_{i,j}$
/* There is a message from controller C_j in the buffer between C_j and C_i */
if $([\tau_i]_i = [\tau]_i) \wedge (Rec = A_i?A_j(m).w)$ **then**
  /* continuation of an already started branch */
  consume (m, τ)
  $numEvt$ - -
  send (j, m) to $K(A_i)$ via port P_i
  $Rec = w$
else
  if $(numEvt = 0) \wedge ([\tau_i]_i \lessdot [\tau]_i)$ **then**
    /* A new branch b was started, and this is the next */
    /* branch that A_i should execute (i is concerned by b)*/
    consume (m, τ)
    $\tau_i := \tau$
    $currentb := b$ s.t. $[\tau][b] - [\tau_i][b] \neq 0$
    $numEvt := |\Pi_i(M_{currentb})| - 1$
    $Rec := tail(\Pi_{i,?}(M_{currentb}))$
    send (j, m) to $K(A_i)$ via port P_i
  end if
  /* The last situation is when the message can not be consumed because it does not have the right sequence number */
end if

Now that we have defined controlled automata and their controllers, we can define formally how they compose. Recall that $K(A_i)$ is a finite state machine with the same states as A_i, but in which each transition $(q, i!j(m), q')$ is replaced by a transition $(q, i!C_i(j, m, b), q')$ (where b denotes the name of the branch currently executed by A_i, and each transition $(q, i?j(m), q')$ is replaced by a transition $(q, i?C_i(j, m), q')$. Each controller C_i is not a communicating automaton, but yet it is a machine that sends and receives messages. The composition $K(A_i) \mid C_i$ of a machine with its controller is a

pair of communicating machines with a FIFO buffer from $K(A_i)$ to C_i, and another from C_i to $K(A_i)$. Then, the composition of controlled machines $\underset{i \in I}{\parallel} (K(A_i)|C_i)$ is the union of all $K(A_i)|C_i$, with communication buffers from each C_i to each C_j, for $i \neq j$ in I. Note that $K(A_i)'s$ communicate only with their controllers. This composition is illustrated in Figure 8, where the depicted architecture is $\big(K(A_i) \mid C_i\big) \parallel \big(K(A_j) \mid C_j\big) \parallel \big(K(A_k) \mid C_k\big)$. At this point, let us note that our controlled implementation is not a CFSM anymore. Note that our controllers are defined with several lines of code, but that they simply recall a local state plus an increasing vector of integers. The number of local states that a controller can record is finite (they are simply the states of the finite automaton obtained by projection on the instance). So, the infinite part of the controller only comes from the vector. Another light modification with respect to standard communicating machines is that the controller needs to read messages without consuming them. Note however, that variables, message reading, etc. are allowed in extended state machine models such as SDL [19]. Considered individually, process descriptions obtained after controlled synthesis are represented by an automaton plus its controller. However, the correctness result presented hereafter shows that the synthesis does not change the individual behavior of an instance, which remains regular. The major difference between the standard architecture and the controlled one is that the controlled automata ‘simulate’ the original specification (controllers are allowed to play additional hidden sequences of events before delivering a message), while the automata obtained by projection in the standard synthesis framework of section 4 have to play exactly the sequences of events described by the original HMSC to be a correct implementation.

5.3 Correctness of controlled synthesis

Let us show correctness of the synthesis with local controllers. Of course, adding controllers to the system means adding the controllers’ actions to the executions. Hence, we can not require that $\mathcal{L}(H) = \mathcal{L}(\underset{i \in I}{\parallel} (K(A_i)|Ci)$ anymore. We propose another notion of correctness, namely language equality up to abstraction of controllers. Abstraction erases controllers’ actions, and considers communications (q, m, b) from a process p to his controller as a communication of a message m from p to q.

Definition 13 *Let $O = (E, \leq, t, \phi, \mu)$ be a prefix in $\mathcal{L}(\underset{i \in I}{\parallel} (K(A_i)|Ci)$. The* restriction *of O to non-control events is a restriction of O to events located on $K(A_i)$'s. We will denote this restriction by $Unc(O)$. The* uncontrolling *of $O = (E, \leq, t, \phi, \mu)$ 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. $Ru(O) = (E, \leq, t', \phi, \mu')$, where $t'(e) = p!q(m)$ if*

$t(e) = K(A_p)!C_p(m, q, c)$, $t'(e) = p?q(m)$ if $t(e) = K(A_p)?C_p(m, q)$, and $t'(e) = t(e)$ otherwise. Function μ' maps the ith sending from p to q with the ith reception on q from p for every pair of processes.

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

Theorem 4 *Let H be an HMSC, and let $\underset{i\in I}{\|} K(A_i)|C_i$ be its controlled synthesis. Then, $Ru(Unc(\mathcal{L}(\underset{i\in I}{\|} K(A_i)|C_i))) = \mathcal{L}(H)$.*

Proof sketch: We want to show that the original specification given as an HMSC and the synthesized controlled machines exhibit the same behaviors. We proceed in several steps. We first show that in the synthesized machines, all choices (i.e. events corresponding to the first event of some bMSC) are causally ordered in any execution of the network of synthesized machines and controllers. We then show that for every configuration of an HMSC H reachable after an execution O, there exists a finite set of configurations of the synthesized machines reachable by observing the same execution. The last steps of the proof show inclusion of specification and implementations languages in both directions by contradiction. Supposing that there exists a configuration of H reached after executing a prefix O that allows firing of an event a but that there exists no corresponding configuration of the CFSM reachable after O that allows a leads to a contradiction. We consider each type of events for a and show that allowing a in one language but not in the other contradicts either the fact that O is a prefix of both the original specification and the synthesized language, or the fact that choices are ordered. A complete proof of this theorem can be found in a research report available at `http://people.rennes.inria.fr/Loic.Helouet/Papers/RR-7597.pdf`□

This result shows correctness of synthesis up to renaming, and erasing of controllers' moves. As a consequence, the behavior of an instance $i \in I$ in an HMSC, and the behavior of the CFSM $K(A_i)$ are isomorphic. Hence, even after adding infinite controllers, the behaviors of processes remains regular.

6 Use case

In order to illustrate the way our algorithm works we study controlled synthesis from an example HMSC describing a simple transmission protocol based on the Morse code. This example is shown in Figure 9. In this figure, we indicate in italic an index associated to each branch of the HMSC (for instance transition $(n_0, M0, n_1)$ is branch $b1$). One can immediately notice that this HMSC is local. However, it is not reconstructible, and the respective ordering between messages *zero* and *one* can be lost. In this example, process A wants to transmit Morse coded information to process B. The protocol can be decomposed into several phases. If A has nothing to send,

it indicates it to the coder which acknowledges that there will be no transmission via a message *Leave* (bMSC M6). To send a message, A requires a connection to B via the Morse Coder C (bMSC $M0$). Once the connection is established A can leave the transmission (bMSC M5), or send the information to the Morse Coder C (bMSC $M1$). Then, C translates the information received from A into a sequence of binary digits (0's and 1's) that represent respectively the dots and the dashes of the Morse code. Then C can send the elements of this sequence to B via two channels: All the 0s are sent via the channel $chan0$ (bMSC $M2$) and all the 1s are sent via the channel $chan1$ (bMSC $M3$). Once the coded information is completely transmitted to B, the process C sends an acknowledgment to the process A that can choose either to send a new piece of data or to close the connection.

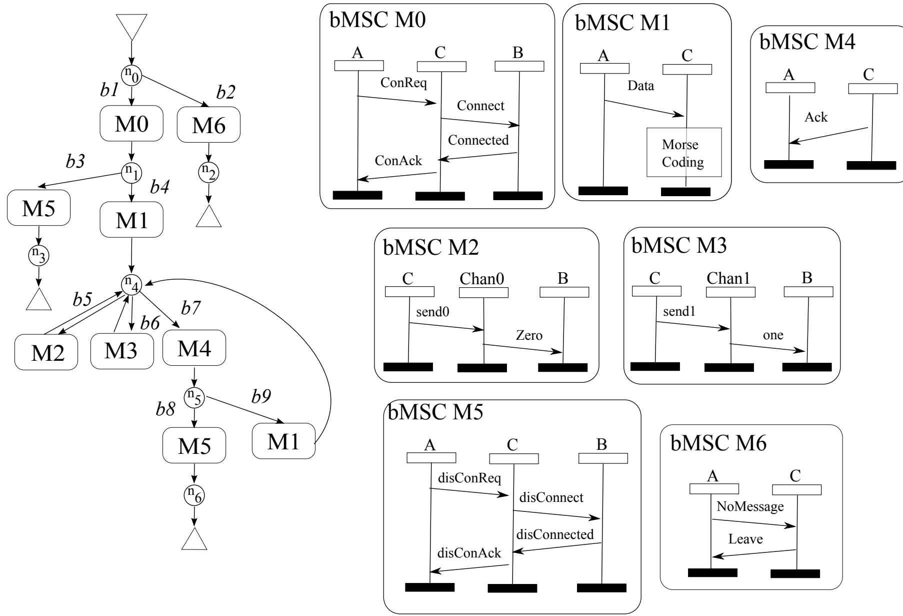

Fig. 9: A simple transmission protocol based on Morse code

Figure 10 shows the projection of the HMSC of Figure 9 on all the instances of the system. As this HMSC is not reconstructible, we must use the controlled synthesis technique described in section 5. Hence, each process interacts with its controller. We will give a number to each process going from 0 to 4 associated respectively to the processes $A, B, C, Chan0$ and $Chan1$, and call $Cont_x$ the controller attached to instance x. Let us explain the structure of the exchanged messages between the automata and their controllers: For example A wants to send to C the message $ConReq$ (hence choosing branch $b1$ in the HMSC), then A sends to its controller the message

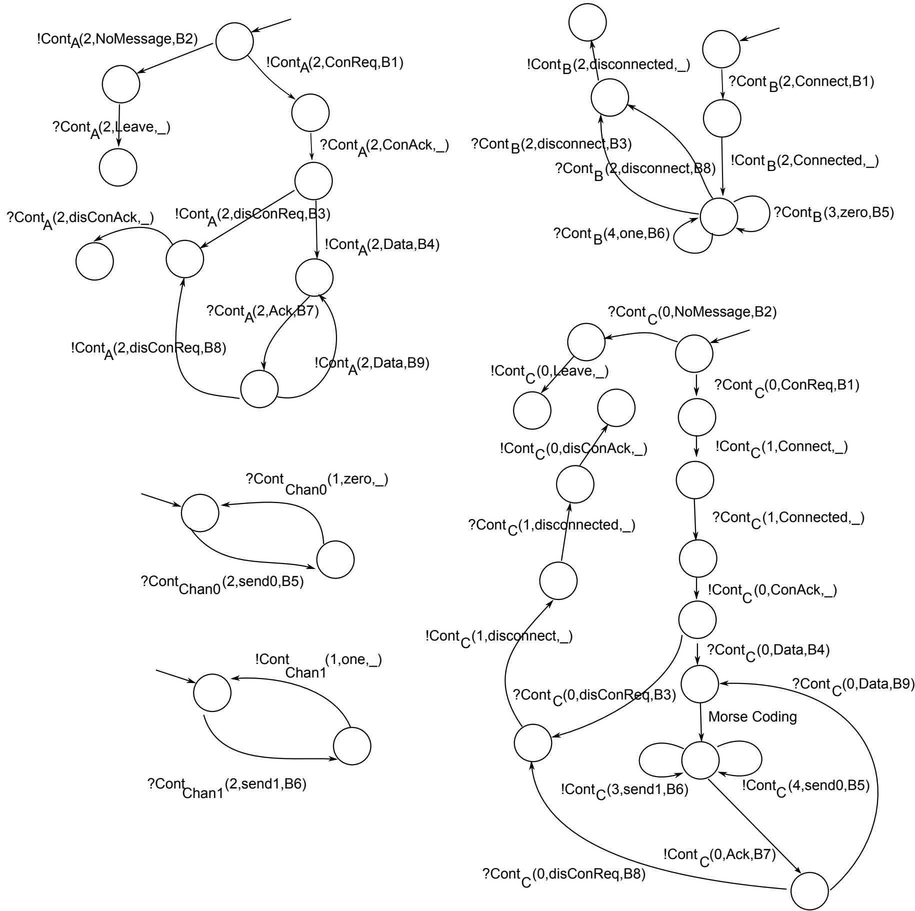

Fig. 10: The CFSM obtained by projection of the HMSC Figure 9

$(2, ConReq, b1)$ that corresponds to the (i, m, j) structure described in the algorithm. The 2 in the message corresponds to the destination process, so when the controller receives the messages it knows to which instance controller it should be forwarded (the process number 2 is C). $ConReq$ represents simply the data part of the message that should be sent by the controller, $b1$ means that the process A wants to start a new scenario (or bMSC). So, upon reception of this message, the controller $Cont_A$ have to update its local tag. The new tag will then be concatenated to the message that will be sent by $Cont_A$. The Morse example can be controlled using tags in $\mathbb{N}^9$, as there are 9 different branches in our HMSC. Note that the tagging mechanisms are not meant to be written by a user, but rather automatically generated by a tool. The tool SOFAT [15], a scenario platform developed at IRISA has been extended to allow such generation.

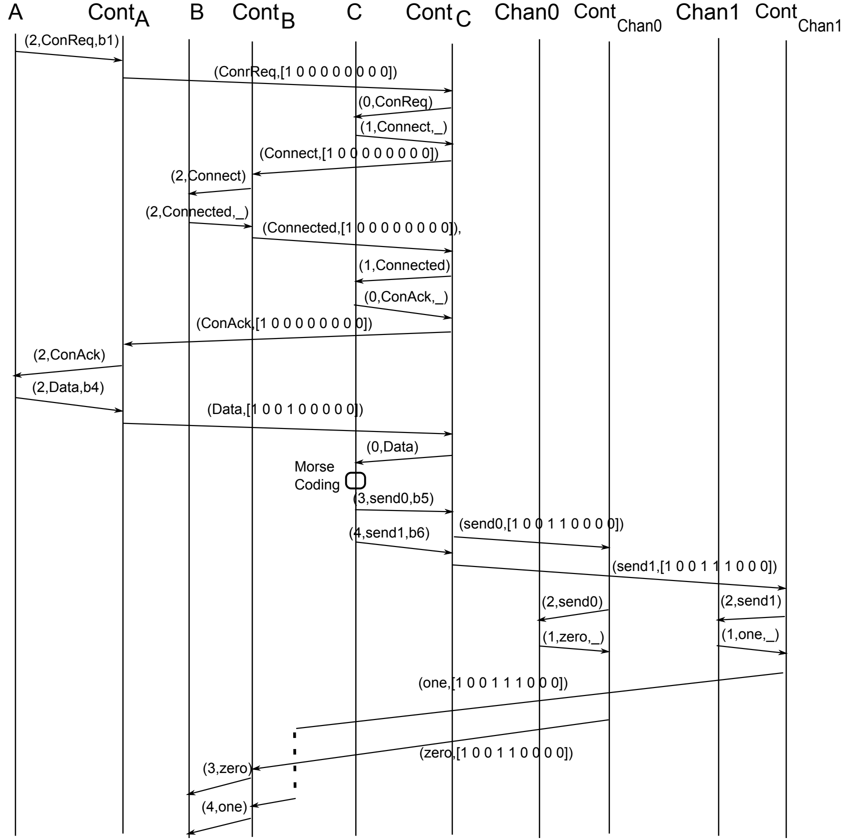

Fig. 11: An execution of the CFSM of Figure 10 with their controllers

An example of execution for the path $n_0 \xrightarrow{M0} n_1 \xrightarrow{M1} n_4 \xrightarrow{M2} n_4 \longrightarrow n_4 \xrightarrow{M3} n_4$ is shown in Figure 11. In $M0$ the connection is established between the processes A and B via C, then process A sends to C the data to be coded and sent to process B. C codes the data: In the example, it is coded as a sequence of two bits: 0 1. Instance C chooses branch $b5$ (bMSC $M2$) then branch $b6$ (bMSC $M3$). A difference in the performance between $chan0$ and $chan1$ causes a delay and a message overtaking in the diagram: The message *one* is received by $Cont_B$ (the controller associated with B) before the message *zero*. Hence, $Cont_B$ has a message of the form (one, τ) in its buffer collecting messages from $Cont_{Chan1}$, and no message in its buffer collecting messages from $Cont_{Chan0}$. Then, $Cont_B$ compares the tag $\tau = [100111000])$ of this message with its local tag $\tau_B = [100000000]$ (set to this value after the execution of the bMSC $M0$ as B is an active instance of $M0$). The projection $[\tau]_B = [10110]$ of the received tag on the branches concerning B is not an immediate successor of $[\tau_B]_B = [10000]$,

so the controller delays the delivery of the message *one* to the automaton B, and leaves the message in its buffer. When $Cont_B$, receives the message *zero* with tag $\tau' = [100110000]$ in its buffer from $Cont_{C0}$, as $[\tau']_B = [10100]$ is the immediate successor of $[\tau_B]_B = [10000]$, the message is forwarded to B and $Cont_B$ updates its local tag to [100110000]. When $Cont_B$ compares again the tag of the delayed message *one* with its local tag, $[\tau]_B = [10110]$ is the immediate successor of $[\tau_B]_B = [10100]$ so $Cont_B$ forwards the message *one* to B and updates its local tab to [100111000]. Hence, a correct ordering in the reception of messages *zero* and *one* on instance B is ensured by the tagging mechanisms despite the message crossing.

7 Related work

Protocol synthesis is not a new problem, and stems from the 80's [7]. The addressed problem is: Given a high level specification with global control (the decisions taken in the system may consider a global state of all agents), how to produce an implementation, i.e. a set of independent (hence with local control only) machines that exhibit the same behaviors as the original specification? This problem has been addressed with many specification and implementation formalisms. For a generic survey on protocol synthesis, we refer interested readers to [28]. Note that synthesis may not be achievable.

In recent years, a lot of attention has been paid to synthesis from scenario notations. Synthesis from scenarios can be easier to implement when the chosen formalism includes indications on communications and on the distribution of tasks among agents. The problem has been addressed for MSCs [1,16], LSCs [14], sequence diagrams [32], and other scenario-based models. We do not claim for exhaustiveness in our bibliography, and refer interested readers to [3,22] for surveys on synthesis from scenarios. These surveys compare and classify many approaches, and we next classify our technique according to the comparison criteria provided in these surveys.

Liang et al [22] compare the synthesis approaches according to the *source formalism*, the *intended use* (analysis or code generation), the *support for composition operators and parallelism*. Our approach uses High-level MSCs as input language, and supports composition operators such as loops, sequence, and choices. We consider parallelism among agents, but there is no support of parallel frames. The reasons for this restriction are discussed in section 2: Parallel frames introduce non-determinism leading to incorrect synthesized behaviors, and may force implementations to have an unbounded number of control states. The *intended use* of our technique is mainly code generation. Other interesting criteria address the *target model*, which can be with global or local control, the *degree of automation*, and *tool support*. The synthesis proposed in this paper derives local finite state machines, which are controlled asynchronously by machines able to delay some messages according to some information about ordering of messages coded as counters. The technique is fully automated, and is implemented in our

SOFAT tool [15]. Last, Liang et al check if the synthesis technique checks *correctness* and *completeness* of the synthesized model. Our synthesis approach is not concerned by these criteria, as the derived CFMS are correct and complete by construction, i.e. they exhibit exactly the same behaviors as the original description.

Amyot et al [3] use some criteria of [22], and introduce several other such as *component focus*, which considers whether the distribution of behaviors is detailed in the specification formalism, *hiding* i.e. the specification formalism considers internal behavior of the modeled system as a black box or allows description of internal details. In addition, Amyot et al consider *representation issues* i.e. whether the specification formalism is graphical or textual, and *ordering issues*, i.e. whether concurrency is made explicit in the formalism, *time* (does the scenario model and the synthesis approach address time issues?), *abstraction* (can the scenario model represent generic behaviors), *identity* (the ability to define generic scenarios involving groups of agents rather than precisely identified ones), and *dynamicity* (the ability to change the behavior of agents at runtime). Clearly, the HMSCs considered in our approach emphasize distribution of actions over agents, and allows for description of internal behaviors using internal actions. HMSCs are both a graphical and textual language. Though the whole HMSC language allows for abstraction, time (use of timers and expression of time constraints on scenarios), decomposition, dynamic process creation, or definition of abstract instances, we do not address these features of the language in our synthesis solutions. The most important and interesting (but also the most difficult) issues to address using such techniques are certainly time and dynamic process creation. However, defining time constraints, for instance can completely change the interpretation of a specification, and even make it inconsistent. Furthermore, time constraints involving events located on distinct instances (for instance the maximal delay allowed between the sending of a message and its reception) are hard to implement. Dynamicity is also hard to address, as there is a lack of formal distributed models allowing dynamic process creation. A first attempt to propose a dynamic communicating automaton model appeared in [8], but the proposed model must be highly non-deterministic in order to implement dynamic MSCs.

Let us now compare our approach with some former synthesis works based on HMSC projection. Most of these works propose solutions for syntactic subclasses of HMSCs only, and usually *local* HMSCs. We have shown in this paper that working with local HMSCs is not sufficient to guarantee a correct synthesis. Indeed, the machines synthesized by the MSC2SDL tool [1] or the MOST tool [23] frequently allow for more behaviors than the original specification. To solve this problem, [16] introduced *reconstructible HMSCs* and showed that synthesis by projection is correct for this subclass. The solution in [12] uses local HMSCs, and furthermore requires that all processes of the HMSC are active (i.e. send or receive a message) in all branches. The approach in [5] considers regular HMSC specifications, that is a subclass of HMSCs with the expressive power of finite automata, and

synthesizes a correct target model. Other works allow the implementation to deadlock [25] and consider that deadlocked runs are not part of the implemented language. In our approach, we have ruled out this possibility, and considered that every initiated run had to be considered as a behavior of the synthesized machines, which seems more realistic. Our approach synthesizes a correct distributed implementation for the whole class of local HMSCs. Correctness is an improvement with respect to [1,23], and completeness an improvement with respect to [12]. Many synthesis approaches proposed these last 10 years assume a synchronous semantics of HMSCs (usually by considering synchronous communications among instances, or synchronization among instances at the end of each bMSC), and take finite state machines, or statecharts variants as target language. The work in [29] assumes synchronous communications in bMSCs, and defines the semantics of HMSCs as a parallel (and synchronous) composition of finite state machines associated to instances. As a result, the synthesized specification is a finite automaton. The work in [21] synthesizes RoomCharts (a variant of statecharts) as target language, and hence assumes a synchronous semantics of HMSCs. The synchronous approach is well adapted to contexts where instances are seen as components of a synchronous system. Synthesizing finite objects then allows for standard model-checking techniques. Again, we refer interested readers to surveys [3,22] for a more exhaustive list of synthesis approaches with statecharts variants as target language. On the other hand, our approach considers the standard asynchronous semantics of HMSCs [18], and allows for the synthesis of independent components, that communicate asynchronously. The semantics of the synthesized machines is not always finite state.

8 Conclusion

We have considered the synthesis problem for HMSCs. Synthesis of CFSMs by a simple projection mechanism is correct for local and reconstructible HMSCs. We have proposed a new solution to synthesize correct implementation for local HMSCs that are not reconstructible: Additional controllers simply tag messages and delay them to ensure correct ordering of message receptions. We think that the class of local HMSCs is a good compromise between the abstraction that is required in a specification formalism, and the preciseness that is needed for a model to be implementable. Indeed, imposing local choices avoids considering in the synthesis some heavy synchronization mechanisms among instances to ensure that distant processes behave according to the same chosen scenario. The class of local HMSCs seems expressive enough to model many interesting protocols, and furthermore, locality of HMSCs is decidable. The synthesis algorithms have been implemented in our tool SOFAT [15], to generate a formal description of the CFSM from an HMSC, Promela code, or even java code for all the instances and controllers needed in the system.

This paper focuses on local HMSCs, but solutions still exist to synthesize correct (up to some abstraction) machines from non-local HMSCs. Indeed, a non-local HMSC can be made local by adding new synchronization messages. In the future, we plan to explore how to integrate the computation of optimal and efficient synchronization in a design methodology. The integration of data is another challenging aspect. The techniques proposed in this paper only address the control flow in a high-level description, and do not consider data. Inserting manipulation of local data in the internal actions of processes can be done easily by mixing the language of bMSCs with a data manipulation language. The code attached to actions can then be copied as it is in the generated code, which does not really impact the synthesis process. However, if data are shared and used to guard choices in HMSCs, the projection technique does not necessarily work, and additional synchronization and consistency mechanisms are needed to ensure that the synthesized processes work with the same data values.

Time issues are also complex to handle. If we consider for instance as an input model a time-constrained MSC [2], synthesizing a correct model means synthesizing machines that meet all the time constraints expressed in the specification. This imposes in particular that controllers should also play the role of timed schedulers. In such a context, using timed languages equality as a notion of correctness for synthesis seems too constraining, and one should probably restrict to timed languages inclusion.

A more technical perspective is to optimize our algorithm to reduce the size of tags. A first challenge is to reduce the number of branches that a controller have to consider. A first intuition is that only non-reconstructible choices should be remembered, but yet this has to be demonstrated. A second possibility is that all branches of a choice need not be remembered if they can not be used as witnesses for non-locality. Another aspect is to try to bound the integers used in choice clocks. This could be done by general decrease of all entries of clocks when every entry has exceeded some threshold k, but with additional synchronization among controllers.

Last, to keep the construction of CFSM simple, we have assumed that communications were FIFO, and as a consequence that the HMSCs considered did not contain message overtaking. The standard HMSCs allow explicit specification of message crossing inside a bMSC (but not of two messages from distinct bMSCs). Extending our techniques to models that allow message overtaking in bMSCs should be easy, as it needs only to allow a bounded number of lookaheads in FIFO buffers. Such possibility was for instance proposed in the MSC2SDL tool [1]. 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 enforce properties such as boundedness of buffers, avoidance of a given configuration, etc.

Acknowledgments: We wish to thank anonymous reviewers for careful reading of this paper, and for numerous comments which helped improving it.

References

1. M. Abdalla, F. Khendek, and G. Butler. New results on deriving SDL specifications from MSCs. In *SDL Forum*, pages 51–66, 1999.
2. S. Akshay, M. Mukund, and N.K. Kumar. Checking coverage for infinite collections of timed scenarios. In *CONCUR'07*, pages 181–196, 2007.
3. D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.
4. D. Amyot and G. Mussbacher. User requirements notation: the first ten years, the next ten years. *Journal of software*, 6(5):747–768, May 2011.
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. G. Bochmann and R. Gotzhein. Deriving protocol specifications from service specifications. In *Proc. of the ACM SIGCOMM conference on Communications architectures & protocols*, pages 148–156, 1986.
8. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *Proc. of CSR (Computer Science in Russia)*, volume 6072 of *LNCS*, pages 48–59. Springer, 2010.
9. D. Brand and P. Zafiropoulo. On communicating finite state machines. Technical Report RZ1053, IBM Zurich Research Lab, 1981.
10. B. Caillaud, P. Darondeau, L. Hélouët, and G. Lesventes. HMSCs en tant que spécifications partielles et leurs complétions dans les réseaux de Petri. Research Report RR-3970, INRIA, 2000.
11. C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.
12. 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.
13. Object Management Group. *UML 2.0 : Unified Modeling Language.* 2005.
14. D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications. *Int. J. Found. Comput. Sci.*, 13(1):5–51, 2002.
15. L. Hélouët, R. Abdallah, and D. Bhatia. SOFAT : Scenario formal analysis toolbox. 2011. `www.irisa.fr/distribcom/Prototypes/SOFAT/`.
16. 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.
17. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

18. ITU-T. Z.120 : Message sequence charts (MSC). Technical report, International Telecommunication Union, 1998.
19. ITU-T. Z.100 : Specification and description language (SDL). Technical report, International Telecommunication Union, 2011.
20. ITU-T. Z.150 : User requirements notation (URN) - language requirements and framework. Technical report, International Telecommunication Union, 2011.
21. S. Leue, L. Mehrmann, and M. Rezai. Synthesizing software architecture descriptions from message sequence chart specifications. In *ASE*, pages 192–195, 1998.
22. H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. of SCESM '06: the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 5–12, 2006.
23. N. Mansurov and D. Zhukov. Automatic synthesis of sdl models in use case methodology. In *SDL Forum*, pages 225–240, 1999.
24. F. Mattern. Time and global states of distributed systems. *in Proc. Int. Workshop on Parallel and Distributed Algorithms , Bonas, France , North Holland*, pages 215–226, 1988.
25. M. Mukund, K.N. Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *CONCUR*, pages 521–535, 2000.
26. A. Muscholl, D. Peled, and Z. Su. Deciding properties for Message Sequence Charts. In *FoSSaCS*, volume 1378 of *LNCS*, pages 226–242, 1998.
27. M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, 1991.
28. K. Saleh. Synthesis of communications protocols: an annotated bibliography. *SIGCOMM Comput. Commun. Rev.*, 26:40–59, 1996.
29. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *ICSE*, pages 188–197, 2001.
30. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in Message Sequence Chart specifications. In *ESEC / SIGSOFT FSE*, pages 74–82, 2001.
31. S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.
32. T. Ziadi, L. Hélouët, and J-M Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *ICSE*, pages 242–251, 2004.