Implementability of distributed systems described with scenarios

Rouwaida Abdallah

Implementability of distributed systems described with scenarios

by

©Rouwaida ABDALLAH

A Thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

PHD in Computer Science

Ecole Normale Superieure de cachan

ENS-cachan

July 2013

Rennes

Abstract

Distributed systems lie at the heart of many modern applications (social networks, web services, etc.). However, developers face many challenges in implementing distributed systems. The major one we focus on is avoiding the erroneous behaviors, that do not appear in the requirements of the distributed system, and that are caused by the concurrency between the entities of this system.

The automatic code generation from requirements of distributed systems remains an old dream. In this thesis, we consider the automatic generation of a skeleton of code covering the interactions between the entities of a distributed system. This allows us to avoid the erroneous behaviors caused by the concurrency. Then, in a later step, this skeleton can be completed by adding and debugging the code that describes the local actions happening on each entity independently from its interactions with the other entities.

The automatic generation that we consider is from a scenario-based specification that formally describes the interactions within informal requirements of a distributed system. We choose High-level Message Sequence Charts (HMSCs for short) as a scenario-based specification for the many advantages that they present: namely the clear graphical and textual representations, and the formal semantics. The code gen-

eration from HMSCs requires an intermediate step which is their transformation into an abstract machine model that describes the local views of the interactions by each entity (A machine representing an entity defines sequences of messages sending and reception). This transformation is called "synthesis". Then, from the abstract machine model, the skeleton's code generation becomes an easy task.

A very intuitive abstract machine model for the synthesis of HMSCs is the Communicating Finite State Machine (CFSMs for short). However, the synthesis from HMSCs into CFSMs may produce programs with more behaviors than described in the specifications in general. We thus restrict then our specifications to a sub-class of HMSCs named "local HMSC". We show that for any local HMSC, behaviors can be preserved by addition of communication controllers that intercept messages to add stamping information before resending them.

We then propose a new technique that we named "localization" to transform an arbitrary HMSC specification into a local HMSC, hence allowing correct synthesis. We show that this transformation can be automated as a constraint optimization problem. The impact of modifications brought to the original specification can be minimized with respect to a cost function.

Finally, we have implemented the synthesis and the localization approaches into an existing tool named SOFAT. We have, in addition, implemented to SOFAT the automatic code generation of a Promela code and a JAVA code for REST based web services from HMSCs.

Table of Contents

Figure 1: Thank you

Chapter 1

Introduction

1.1 Context

A distributed system consists of a collection of autonomous entities (i.e., computers, processes), that are connected through a network which enables them to communicate and to share common resources. From 1945 until mid 80's, computers were large and expensive: A mainframe used to cost millions of dollars; even minicomputers used to cost at least tens of thousands of dollars each. That is why, most organizations only had a handful of computers. Furthermore, these computers operated independently because there was no way to connect them. Since the mid 80's, the advances in technology, namely the development of powerful microprocessors and the invention of high-speed networks, have begun to change that reality [80]. Since, distributed systems have become widely used in many applications that range from television sets and train signaling systems to e-commerce and stand-alone PC-based software applications. These days distributed systems have become a need, as many recent applications are by nature distributed (bank teller machines, airline reservations, ticket purchasing, communication applications, social networks, etc.).

1.2 Motivation

Nowadays, distributed systems are everywhere and there is a concrete need for implementing functional, usable, and high-performance distributed systems. It is therefore

important for the developers to have an understanding of the requirements of the system and the problems that may occur. Actually, the various entities in a distributed system can operate concurrently and possibly autonomously and this concurrency gives rise to a number of well-studied problems: Processes may use old data; they can make inconsistent updates; the order of updates may or may not matter; the system might deadlock; the data in different systems might never converge to consistent values; etc [44]. Several of these problems come from the erroneous behaviors that occur in the system and that were not described in the requirements. Actually, it is not an easy task to correctly move from requirements towards a distributed implementation while preserving the set of required behaviors for the entities of the distributed system. We have mainly two distinct approaches to go from requirements to implementation:

On one hand, developers consider generally to go directly from informal requirements to implementation. Prototyping and testing remain the principal methods for developers for exploring designs and validating implementations. Methods like the V-Model software development process, presented in Figure 1.1, may be used in the development of distributed system. The V-Model consists respectively in defining a design that describes the requirements of the system, implementing the corresponding code, and finally testing this code. The V-Model software development process might be repeated several times while still finding modifications to do. Such methods are expensive in terms of time and money (the code might be tested and modified several times) and provide only partial coverage of the range of behaviors that a piece of software may exhibit (it is hard to cover and test all the behaviors that may occur on the system, and when the set of behaviors is infinite testing all of them is impossible).

On the other hand, the second approach, which is the one that we consider in this thesis, is the implementation based on the use of scenario-based specifications. Scenario-based specifications present the abstract descriptions of the interactions between the entities of the system. They have become popular as a powerful means of communication for system requirements due to their simplicity and expressive power [40]. Some scenario-based specifications have solid mathematical foundations that can be used to support rigorous analysis and mechanical verification of properties. They allow verifications of system requirements at early stages before the implementation of the code of the system. Their use ranges from requirements engineering [40] and formal specifications [75] to code synthesis [3] and test case specification and generation (e.g. [32]). Furthermore, scenario-based specifications are used to move from requirements towards a skeleton of an implementation for distributed systems and thus to facilitate

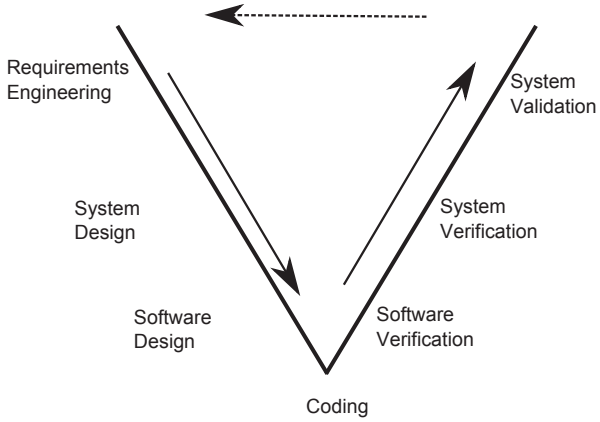


Figure 1.1: The V life cycle model in software development

their construction: Scenario-based specifications mainly describe the interactions between the entities of the distributed system and not the detailed behaviors occurring locally on each entity. Then, the implementation of scenarios results in a skeleton of code that presents the interactions within the distributed system described in the requirements. The rest of the code, that describes the local behaviors for each entity, can then be debugged and added to the skeleton of code to get the complete implementation of the distributed system. Furthermore, the automatic generation of this skeleton of code is very important and offers many advantages:

- The errors that might be induced by developers' implementations are avoided, as this transformation from scenario-based specification into a code is an error-prone task. The correct generation of the skeleton helps the developers to avoid the problems caused by the concurrency in distributed systems.
- Skeleton's code generation is time saving and can lead to a relatively fast generation of prototype and test cases software.
- The high redundancy in distributed systems' code, makes this automatic generation a desirable goal. It will save time needed for writing similar and redundant code.

In this thesis, we are interested in producing a reliable implementation for distributed systems. We will consider a scenario-based specifications approach and our main target is to propose a method that transforms requirements into a skeleton of code that guarantees correct interactions behaviors in a distributed system.

1.3 The synthesis problem

To proceed the skeleton's code generation from a scenario-based specification, we have an intermediate step. This step transforms the scenario-based specification, which is a high-level specification that describes the behaviors of the system from a global point of view, into an abstract machine model that describes the local views (the communicating machines, which define sequences of messages sendings and receptions) that is consistent with the original specification. Then, from the abstract machine model the skeleton's code generation becomes an easy task. This transformation from a high-level specification to an abstract machine model is called the *synthesis*.

This thesis addresses the automatic synthesis problem in the context of distributed applications running on networks of computers, and more precisely correct synthesis algorithms. Synthesis is correct when the abstract machine model preserves the behaviors described in the high-level specification.

In the literature, many different definitions of scenario-based specifications can be found [23, 19, 45, 50]. There are significant differences in terms of syntax, features, semantics, etc. (a more detailed presentation and comparison of scenarios is presented in chapter 2). Sequence charts are one of the approaches to describe scenarios. Sequence charts have been used to describe system behaviors for some time before the International Telecommunications Union (ITU), has undertaken their standardization process. This has resulted in a language called Message Sequence Charts (MSCs). MSCs have undergone several revisions since their first version, the latest one being in 2011 [1].

MSCs are particularly useful in the early stages of system development; they allow describing the communications of a system and can be used to find design errors. First, the graphical representation of MSCs is one of the reasons for their popularity. It makes MSCs intuitively comprehensible and easy to learn and there is no need to have a mathematical background to start using this notation. Furthermore, MSCs have a textual representation that was originally intended for exchanging MSCs between tools. Last but not least, MSCs also have a formal semantics, which allows them to be used for various analysis purposes. Since MSCs are used at a very early stage of design, any error revealed during their analysis yields a high pay-off. This has already motivated the development of algorithms for a variety of analyses including the presence of a race condition in an MSC [9], model checking [10], pattern

matching [69], detection of non-local choices [15, 31], deadlocks, livelocks, and many more (for more details see e.g. [24]).

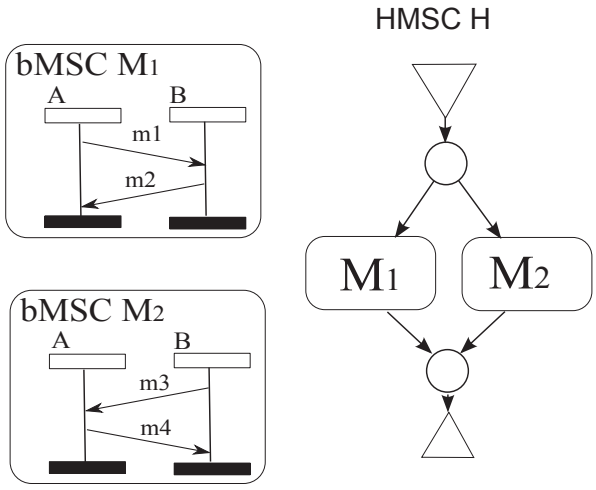


Figure 1.2: Example of MSCs: two bMSCs M_1 and M_2 and one HMSC H

MSCs are composed of several specification layers. At the lowest level, basic MSCs (bMSCs for short) defining finite specifications of interactions among processes. For instance, Figure 1.2 shows two examples of bMSCs M_1 and M_2, where two processes A and B interchange messages: In bMSC M_1, the process A sends a message $m1$ to the process B then B sends a message $m2$ to A. M_2 presents another scenario where B sends the message $m3$ to A then A sends $m4$ to B. However, the real systems are often very complex. MSC specification allows addressing the complexity of distributed systems by composing bMSCs with several means. High-level MSCs (HMSCs for short) describe the composition of bMSCs in a clear and attractive way, which makes them the most used composition mechanism. For instance, Figure 1.2 shows an example of an HMSC H, that composes the two bMSCs M_1 and M_2. It describes an alternative between the bMSC M_1 or M_2. The whole MSC formalism will be described later in chapter 2.

In this thesis, we will consider the synthesis of HMSCs. A very natural way to synthesize abstract machine models from HMSCs is by projection (see chapter 3). The principle of projection is to copy the original behaviors specified in the HMSC specification on each process in the distributed system, and to remove the part of the behaviors that do not belong to this considered process.

A very intuitive abstract machine model for the projection of HMSCs is the Communicating Finite State Machine [17] (CFSMs for short). This model presents several advantages; it allows the definition of concurrent components 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, ...).

Unfortunately, all the global coordination expressed by HMSCs cannot always be translated to CFSMs in the synthesis by projection algorithm. Consequently, some HMSC specifications may not be implementable as CFSMs.

For instance, HMSCs allow for the definition of distributed choices that are 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 an unspecified situation occurs, the implementation is not always consistent with the original HMSC: It exhibits more behaviors and even worse, the synthesized machines can deadlock. For instance, in the HMSC H of Figure 1.2, the process B can send the message $m3$ to A, and at the same time A might send $m1$ to B. In this case, we have more behaviors than what is defined in H where only one bMSC can be run. The processes A and B will deadlock because A considers that it is running the bMSC M_1 then after sending the message $m1$ it will wait for $m2$ from B. On the other hand, B considers that it is running the bMSC M_2 and will wait for $m4$ from A. We consider that correct implementations should not deadlock. HMSCs that do not contain distributed choices are called *local HMSCs*, and are considered as a reasonable sub-class to target a distributed implementation. However, the deadlock-free synthesis solutions proposed so far (see chapter 2 for more details) do not apply to the whole class of local HMSCs.

1.4 Contribution

1.4.1 Any local HMSC is correctly implementable

In this thesis, we first propose a new implementation mechanism that applies without deadlocks to the whole class of local HMSCs, that is a class of HMSCs that do

not require distributed consensus to be executed. The proposed synthesis technique is to project an HMSC on each process participating to the specification. However, even the projection of local HMSCs may produce programs with more behaviors than described in the specification because the order between two consecutive choices can be lost. That is why we compose the projections with local 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 first part of the thesis is that the projection of the behaviors of the controlled system on behaviors of the original processes is equivalent (up to a renaming) to the behaviors of the original local HMSC.

1.4.2 The synthesis of non-local HMSCs

Non-local HMSCs are generally considered as too incomplete or too abstract to be implemented. Therefore, we extend synthesis to general HMSCs by proposing a localization procedure that transforms any non-local HMSC into a local one, and thus allowing its synthesis into CFSMs. The localization can be achieved by adding new messages and processes in scenarios. We have an infinite number of solutions for the localization problem but we are interested in finding solutions with the minimal number of added messages because they correspond to the less disturbing transformation of the specification. We propose to address the localization problem with a constraint optimization technique that finds the best way to add processes and messages in an HMSC specification to transform it into a local HMSC. The experiments we ran on a large class of randomly generated HMSCs, with a prototype tool implementation, show that the localization problem can be solved in general in a few seconds on ordinary machines.

1.4.3 SOFAT tool

We have implemented the proposed approaches into an existing tool called SOFAT (Scenario Oracle and Formal Analysis Toolbox). SOFAT is a formal toolbox for the manipulation of scenarios. SOFAT provides several functionalities, like: syntactical analysis of scenario descriptions, formal analysis of scenario properties, and many others. In this thesis, we have extended SOFAT with synthesis approach for local HMSCs: First we added the automatic generation of CFSMs from local HMSCs. Then from this model, we added the automatic generation of a Promela code(allowing

the verification of some MSCs properties using the XSPIN tool), and JAVA code for REST based web services. We have also implemented the localization procedure as well into SOFAT, so we can get the optimal way to transform any non local HMSC into a local one.

1.5 Thesis organization

This thesis is organized as follows: In chapter 2, we present a brief state of the art on scenario-based specifications in general and Message Sequence Charts in particular. We also present some abstract machines models and in particular the Communicating Finite State Machines model. Then we analyze some important works on synthesis from MSCs and some of the existing sub-classes of MSCs. In Chapter 3, we propose a solution based on local control and message tagging to implement correctly local HMSCs. In Chapter 4, we propose an encoding of minimal localization as a constraint optimization problem, and show the correctness of the approach. In addition, we describe an experimentation conducted to evaluate the performance of our localization procedure, and comment the results. Chapter 5 presents a prototype called SOFAT. We mainly present the functionalities that we have added namely: the projection of an HMSC into a CFSM, the code generation of Promela and JAVA code for a REST platform. Finally, we conclude and present some perspectives in Chapter 6.

Chapter 2

State of the art

This chapter presents the basic definitions and formalisms concerning MSCs, some of their variants, and the synthesis problem.

Message Sequence Chart (MSC for short) is a partial-order based formalism standardized by the International Telecommunication Union [35]. Basically, an MSC describes the communication behavior of a number of logically or physically distributed processes that run in parallel and communicate by exchanging *asynchronous* messages. MSCs and their variants are widely used to capture use cases and requirements during the early design stages of distributed systems. They have been adopted within several software engineering methodologies and tools for concurrent, reactive and real-time systems. e.g. [6, 9, 79], and a variant called Sequence Diagrams has been integrated to UML 2.0 (see [71]).

MSCs are composed of several specification layers. At the lowest level, basic MSCs define interactions among instances, and then these interactions are composed by means of High-level MSCs (HMSCs for short).

2.1 Basic Message Sequence Charts

Basic Message Sequence Chart is the core language of MSC. A bMSC defines a simple scenario describing the communication behaviors and the internal actions of a finite set of entities (called *instances*) in a distributed software system.

2.1.1 Graphical representation

Graphically, a bMSC is presented by a frame containing a graphical representation of the instances. Instances are referred to by means of their names, so these must be unique within a bMSC. An instance is represented by a vertical axis (a top down progressing time line) where events are ordered. The axis starts with the *instance head* symbol (white rectangle) and ends with the *instance end* symbol (black rectangle). The two symbols do not describe the creation and the termination of the instance, but the start and the end of the behaviors of the instance in the description. Figure 2.1 from [38] summarizes the different kinds of events that can be found within a bMSC.

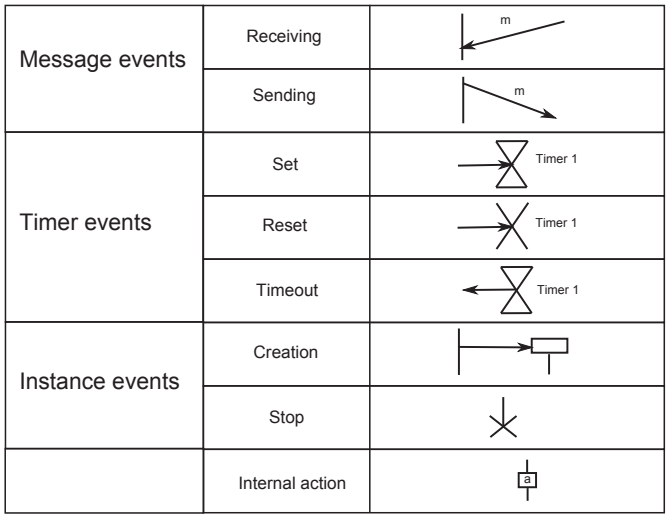


Figure 2.1: Types of events in a bMSC

The message exchanges are represented by arrows labeled by a message name. The local actions are denoted by boxes labeled with the name of the action. Figure 2.2 presents a simple example of a bMSC named $bMSC\_Example$. This example describes a scenario involving three instances $\{Sender, Medium, Receiver\}$ described by three vertical axes. The arrows labeled $\{Data, Ack, Info\}$ between the instances describe messages that are exchanged. The box labeled by a denotes internal activity of instance *Sender*. The event $et1$ on the instance Sender is to start a timer for 10 units of time and the event $et2$ is the timeout of the timer. The timer means that *Sender* should receive the message *Ack* before the timeout of the timer.

As it is presented in Figure 2.1, bMSCs allow for the creation and the termination of processes and the time handling. Time handling and conditions are also supported

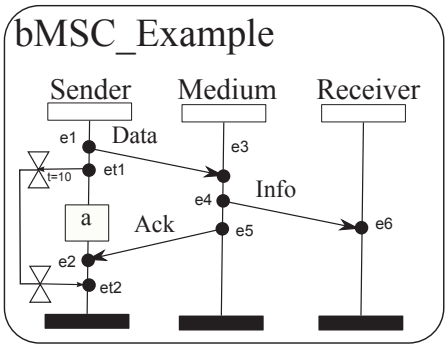


Figure 2.2: An example of bMSC

in bMSC specifications. They are used to improve the readability of the bMSCs, but they do not have any specific semantic meaning. Figure 2.3 shows a condition labeled "Data Processed" that concerns to the two instances *Sender* and *Receiver*. As one

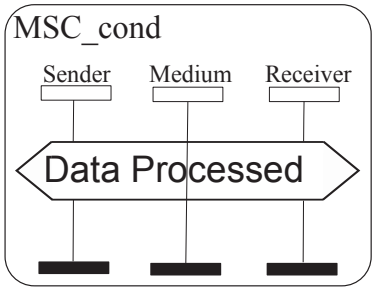


Figure 2.3: Condition Example

can see, the graphical representation of bMSCs is rather intuitive.

2.1.2 Textual representation

BMSCs also have a textual representation that was mainly intended to be an exchange formalism for case tools using MSCs (Telelogic Tau[42], Object Geode [86]). The following example describes the bMSC presented in Figure 2.2:

msc *bMSC_Example*
  instance *Sender* ;
    out *Data* **to** *Medium* ;
    action *a* ;
    in *Ack* **from** *Medium* ;
  endinstance ;
  instance *Medium* ;
    in *Data* **from** *Sender* ;
    out *Info* **to** *Receiver* ;
    out *Ack* **to** *Sender* ;
  endinstance ;
  instance *Receiver* ;
    in *Info* **from** *Medium* ;
  endinstance ;
endmsc ;

In the previous example, the two keywords **msc** and **endmsc** delimits the bMSC description and, in between, come the name of the bMSC and the description of the instances. Each instance's description is delimited by the two keywords **instance** and **endinstance** and, in between, come the name of the instance and the description of the events that occur on this instance ordered by their occurrence time. A message output event is described by: **out** *mssg* **to** *d*, where *mssg* is the name of the message and *d* its receiving instance. In the same way, a message input is described by: **in** *mssg* **from** *s*, where *mssg* is the message name and *s* its sending instance. A local action *a* is described by : **action** *a*. In this example, the instances are presented in the order of their representation in the bMSC, but this is not required by the recommandation Z.120.

2.1.3 Formal definition

A bMSC can be defined formally as follows:

Definition 2.1.1 (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 instance in Figure 2.2, we have $\mu(e1) = e3$ (where $t(e1) = Sender!Medium(Data)$ and $t(e3) = Medium?Sender(Data)$), thus $e1$ and $e3$ are ordered as follows: $e1 \leq e3$. On the other side, events $e3, e4$ and $e5$ occur on the same instance $Medium$, so they are ordered as well, and we have: $e3 \leq e4$ and $e4 \leq e5$.
The semantics of a bMSC M is given in terms of sequences of actions allowed by the causal ordering $\leq$. More formally, we have:

Definition 2.1.2. *A linearization of a bMSC M is a word $w = a_1....a_{|E|}$ that is the labeling of some linear extension of M (i.e. a total order on E respecting the causal ordering $\leq$). The semantics of M is the set of all its linearizations, and is denoted $Lin(M)$.*

2.1.4 Events Ordering and Co-regions

A bMSC defines a precedence relation between events:

- the sending of a message precedes its reception
- all the events specified on the same instance are causally ordered. This order on the axis can be relaxed in some parts of the instance called co-regions.

A co-region is represented graphically by dashed parts of the instance axis(see Figure 2.4). Events specified within a co-region are not necessarily concurrent: Their order is not specified yet, or is not important for the specification. For instance, in Figure 2.4 we have $s1 \leq s2$ and $s1 \leq s3$. The two events $s2$ and $s3$ are in a co-region then we have $(s2 \leq s3) \vee (s3 \leq s2)$.

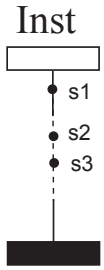


Figure 2.4: Graphical representation of coregion

2.1.5 Gates

MSC also allow messages called gates which describe messages coming from some other instances not described in the MSC. Gates could be regarded as being a simple way to model the passing of data between a sequence diagram and its environment. A gate is depicted as message arrow connected to surrounding frame of the bMSC where the name of gate is presented. Example of gates' usage is shown in Figure 2.5. Other

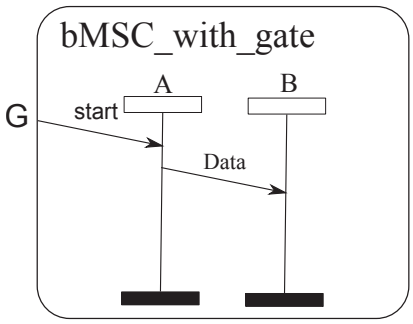


Figure 2.5: Example of bMSC with gate

basic concepts about bMSCs can be found in its ITU standardization documents [46].

2.1.6 MSC composition

The bMSCs present very simple and finite specifications, however, the real systems are often very complex. Many specification languages offer a way to address this complexity. MSC specification allows several type of composition described using composition operators. MSC specifications provide, mainly, three types of composition [75]:

- The sequential composition using the operator *seq*: Composing sequentially two bMSCs $M1$ and $M2$ results in a bMSC where events of the first bMSC $M1$ on each instance $p \in I$ occur before the events of the second bMSC $M2$ on instance p.
- The parallel composition with the operator *par*: When two MSCs are composed with parallel composition the events on the common instances are interleaved. This can be expressed in a coregion as shown in Figure 2.6. In the case that the bMSCs have no common instances, the composition is similar to the sequential composition.
- The alternative composition using the operator *alt*: In most of the systems we might, at some points, have several possible behaviors.

Figure 2.6 presents two bMSCs called respectively $First\_bMSC$ and $Second\_bMSC$. The sequential composition of these two bMSCs gives the bMSC V_bMSC, and their parallel composition gives the bMSC P_bMSC. The expression $First\_bMSC$ *alt* $Second\_bMSC$ means that either $First\_bMSC$ is executed or $Second\_bMSC$ is executed.

In addition to the seq, alt, par operators, MSC allows the following constructs:

- $High - level\ message\ sequence\ charts$ ($HMSCs$): Here the composition is described in a automaton like format. they will be described in details in the next section.
- $MSC\ references$: MSC references can be used inside an MSC (bMSC or HMSC) to refer to another one. An example of the use of MSC references is presented in Figure 2.7. In this example, the bMSC ref_bMSC contains an MSC reference expression that is attached to the instances A, B, and C and that contains the expression $First\_bMSC$ *alt* $Second\_bMSC$. This means that we execute either the bMSC $First\_bMSC$ or the bMSC $Second\_bMSC$ then the instance C sends the message md to D.

- *Inline expressions*: These expressions allow the description of the composition of MSCs within an MSC. Figure 2.8 shows an example of the graphical representation of an inline expression. At the beginning the process K has the choice between sending the message m or the message n.

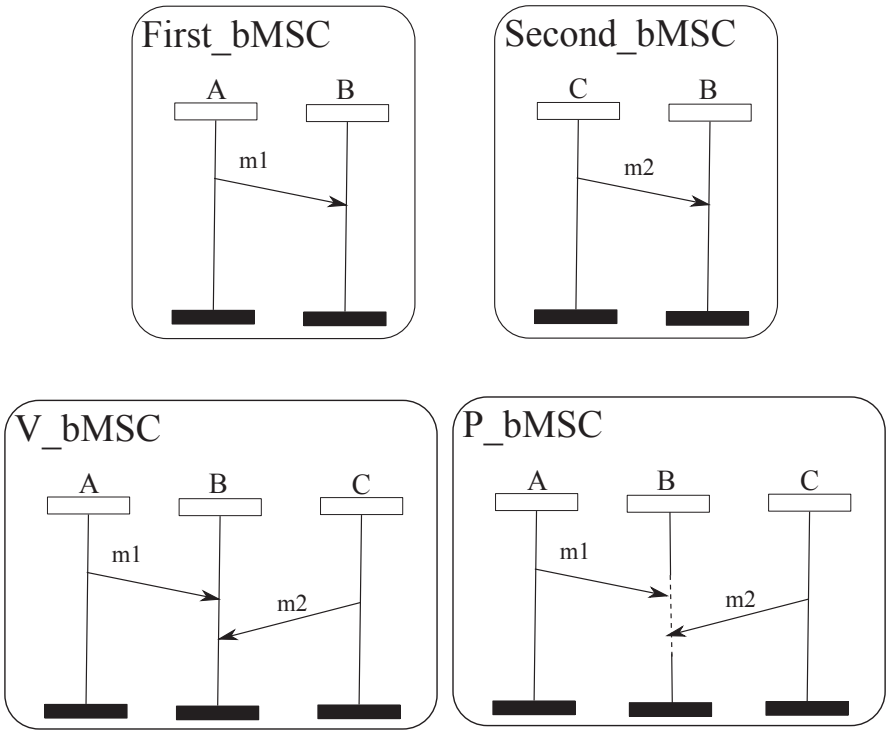


Figure 2.6: Sequential and parallel composition

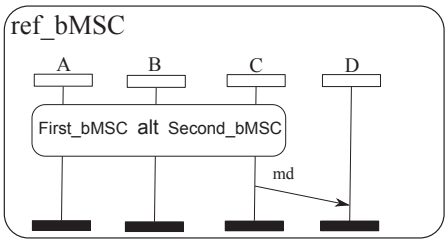


Figure 2.7: Example of MSC Reference Expressions

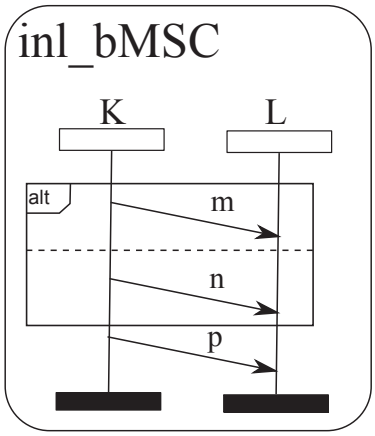


Figure 2.8: Example on MSC Inline Expressions

2.2 High-level Message Sequence Charts

High-level Message Sequence Charts [35] describe the composition of MSCs in a clear and attractive way, which makes it the most used composition mechanism. HMSCs allow to represent the composition situations covered by *MSC references* and the *Inline expressions* as well.

2.2.1 Graphical representation

Graphically an HMSC is represented as a directed graph which nodes are of the form presented in Figure 2.9. A *reference symbol* can contain a reference to a bMSC, another HMSC, or any other *MSC reference expression*. A condition symbol may contain one or several condition names. The *start* symbol and the *end* symbol are respectively the initial and the terminal nodes of an HMSC.

Every HMSC should have exactly one *start* symbol and the graph must be connected so that any node can be reached from the *start* node. In the directed graph, all nodes have outgoing arrows except an *end* node, and all have incoming arrows except the *start* node.

The connection nodes (called choice nodes) are used mainly when we have several possible choices or alternatives at a point of the HMSC, and they are used as well to connect other nodes to make the HMSC easily readable. A choice node that is directly connected to the *start* node is called *initial* node, and the one directly connected to

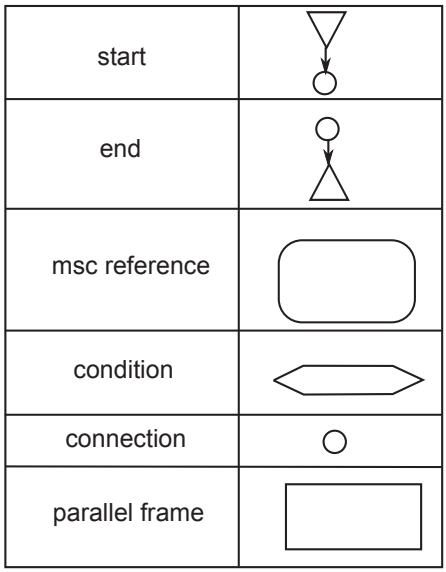


Figure 2.9: HMSC symbols types

the end node is called *sink* node. Figure 2.10 presents a first 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.

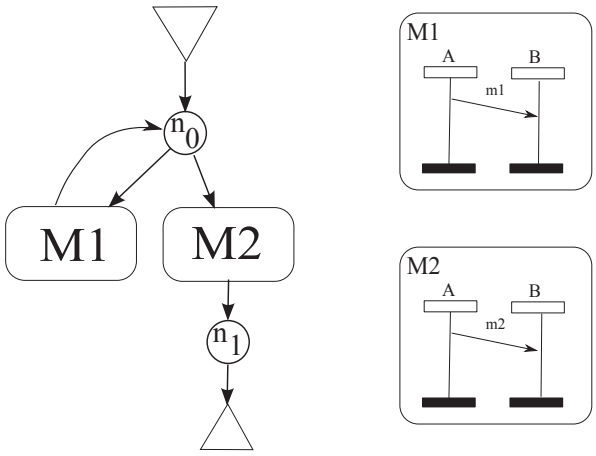


Figure 2.10: An example of High-level Message Sequence Chart.

A *parallel frame* denotes the parallel composition of one or several HMSCs that it contains. Figure 2.11 shows the graphical representation of a parallel frame.

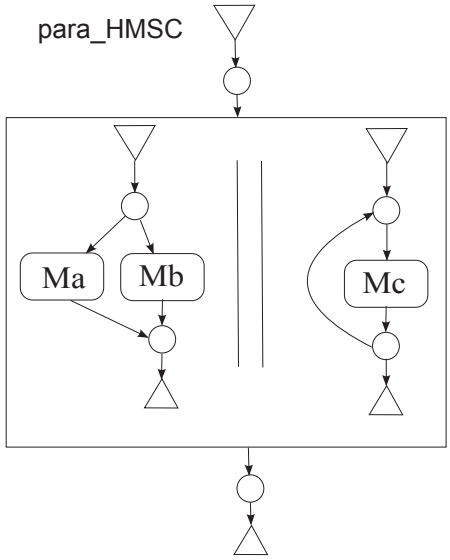


Figure 2.11: An example of HMSC with a parallel frame.

2.2.2 Textual representation

As for bMSCs, HMSCs have a textual description. The HMSC from Figure 2.10 is presented textually by:

msc *Example* **;**
expr l0 **;**
l1: **connect seq** (l2 **alt** l3) **;**
l2: M1 **seq** (l1) **;**
l3: M2 **seq** (l4) **;**
l4: **connect seq** (l5) **;**
l5: **end ;**
endmsc ;

2.2.3 Formal definition

In this thesis we will consider HMSCs without co-regions or parallel frames. This choice is argued in chapter 3 section 3.1.4. The formal definition of the HMSC that we consider can be presented as follows:

Definition 2.2.1 (HMSCs)**.** *An* HMSC *is a graph* $H = (I, N, \rightarrow, \mathcal{M}, n_0, Fin)$*, where*

- I *is a finite set of instances,*

- *N is a finite set of nodes, $n_0 \in N$ is the initial node of H, and $Fin \subseteq N$ is the set of final states,*
- *$\mathcal{M}$ 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.*

In the example of Figure 2.10, $\mathcal{M} = \{M_1, M_2\}$ and the transition relation contains two transitions, namely (n_0, M_1, n_0) and (n_0, M_2, n_1). The behavior M_1 can be repeated an arbitrary number of times, and then be followed by the behavior described in M_2. we would like to mention that running M_1 followed by M_2 does not mean that all the events described by M_1 occur before the ones described by M_2 (the process B might receive the message $m2$ before receiving $m1$).

Before giving the definitions of the semantics of HMSC let us formally present Sequential composition of two bMSCs. 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 HMSCs, that compose sequentially bMSCs to obtain *languages of bMSCs*. The sequential composition is formally defined as follows:

Definition 2.2.2 (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 the bMSC $M_1 \circ M_2 = (E, \leq, C, \phi, t, \mu)$, 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 chapter, 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 2.12: 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.

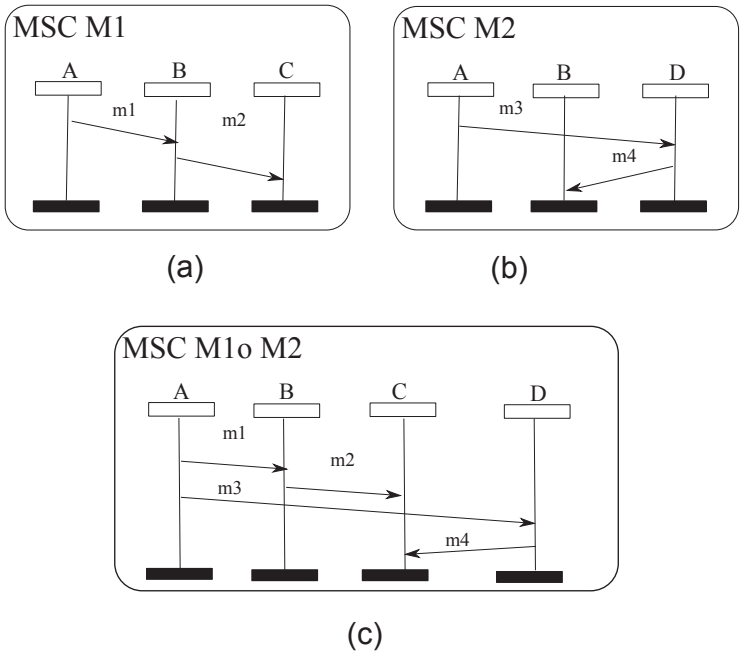


Figure 2.12: Two bMSCs M_1 and M_2 and their concatenation $M_1 \circ M_2$

Definition 2.2.3 (HMSC behavior)**.** *Let $H = (I, N, \rightarrow, \mathcal{M}, n_0, Fin)$ be an HMSC. A* path *of H is a sequence $\rho = (n_0, M_0, n_1)(n_1, M_1, n_2)\ldots(n_k, M_k, n_{k+1})$ 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 $\rho = (n_0, M_0, n_1)\ldots(n_k, M_k, n_{k+1})$ in $Paths(H)$ defines a sequence $M_0.M_1\ldots M_k \in \mathcal{M}^*$ of bMSCs. We will denote by M_ρ the bMSC associated to ρ and define it as $M_\rho = M_0 \circ M_1 \circ \cdots \circ M_k$.*

The semantics of an HMSC is given both in terms of generated MSCs $\mathcal{F}_H = \{\ M_\rho \mid \rho \in Paths(H)\ \}$, and linearization $L_H = \{\ \mathrm{lin}(M_\rho) \mid \rho \in Paths(H)\ \}$.

2.3 MSCs Versions

Message Sequence Charts emerged from the SDL (ITU-T Specification and Description Language) community leading to its first ITU-T recommendation in 1992 . Later there have been revisions of MSC in 1996 [77], in 2000 [35], in 2004 [46], and more recently in 2011 [1]. MSC 2000 differs from MSC-96 mainly in the following areas: better integration of conditions, quantitative notion of time, data specification. We refer the reader to [35] for further details. So far, there exists no formal semantics comparable to the one of MSC-96 for MSC-2000. MSC-2004 is a natural continuation of the MSC-2000 version, refining concepts including: extended data interface, and references to default SDL interface, uni-directional time constraints, and in-line high level expressions. The MSC 2011 is intended to be the same as the 2004 it is only correcting a number of errors into the main text and the appendix. Figure 2.13 shows a brief history of the evolution of the MSC standard.

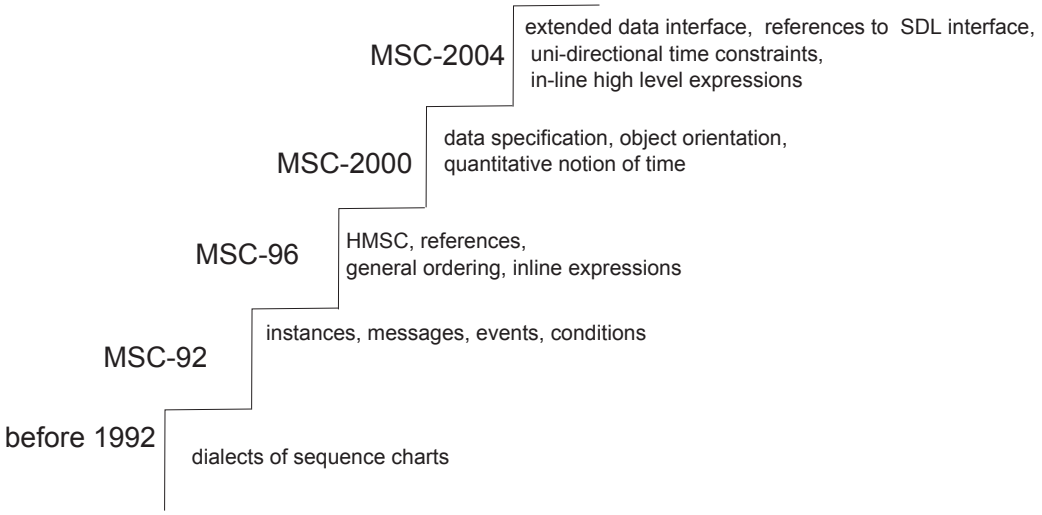


Figure 2.13: Brief history of MSCs.

2.4 Variants of Message Sequence Charts and similar notations

Several variants of MSCs exist [55]. We present some of the most popular ones in this section.

2.4.1 Interworkings

Interworkings is a graphical formalism for describing communications between components of a system. It also has a formal semantics based on process-algebra [63]. Interworkings were first developed to be used in the analysis phase of the developement process at PKI (Philips Kommunikations Industrie) Nürnberg for the message interactions between blocks [50]. They were also used in the specification of radio communication systems and other industry telecommunication applications. Interworkings are considered as one of the direct predecessors of MSC-96. However, they can only model synchronous communications which means that messages receptions cannot be delayed as in MSCs, where the communications are asynchronous. Several elements from MSC-96 and other recent versions are absent from Interworkings like asynchronous messages, gates, instance creation and stop, timers, etc. In particular, there is no means for expressing alternatives and repetition, and no referencing mechanism.

2.4.2 UML Sequence Diagram

The *UML Sequence Diagrams* (SDs for short) are one of the UML diagrams to model the dynamics of a system. Originally, they result from two modeling diagrams: Ivar Jacobson's interaction diagrams[45], and an Object Oriented variant of MSC-92 language called OMSC[19]. SDs are very popular for their role within use case driven object oriented software engineering. They are used to describe either the interactions between the system and the actors of its environment or the communications between objects in a system. However, the SDs are not as formal as MSCs. In [78], the authors propose to make a harmonization between the UML SDs and the MSCs so that they have a mutual benefit: MSCs benefit from the popularity of SDs and SDs benefit from all the advantages that offer the MSCs (mainly the composition mechanisms). [34] consider that a specific MSC profile of UML 2.0 could add the innovative data mechanism which possibly could make it easier to handle Interactions formally.

2.4.3 Live Sequence Charts

Live Sequence Charts (LSCs for short)[23] is a language for scenarios, based on bMSCs. LSCs provide the means to distinguish mandatory and provisional behaviors during system runs. In [23], the authors relate LSC specifications to system runs. A system run, in their approach, is an infinite sequence of snapshots, where a snapshot

consists of the set of current events (being either synchronous or asynchronous sends or receives between components or between a component and the environment), and an assignment of values to all variables of the system.

LSCs provide the means to distinguish mandatory and provisional behaviors on the level of the whole chart and three other elements: messages, locations and conditions. This distinction is achieved graphically by using solid line for mandatory LSC element and dashed lines for possible ones. Table 2.14 summarizes the dual mandatory/provisional notions supported in LSCs, with their informal meaning: Mandatory charts are classified as Universal LSC and provisional ones as Existential LSC. The distinction regarding an internal chart element is referred to as the element's *temperature*; mandatory elements are *hot* and provisional elements are *cold*.

		Mandatory	Provisional
Chart	Mode Semantics	*Universal* All runs of the system satisfy the chart	*Existential* At least one run of the system satisfies the chart
Location	Temperature Semantics	*Hot* Instance run must move beyond location	*Cold* Instance run need not move beyond location
Message	Temperature Semantics	*Hot* If message is sent it will be received	*Cold* Receipt of message is not guaranteed
Condition	Temperature Semantics	Hot Condition must be met; otherwise abort	*Cold* If condition not met exit current (sub)chart

Figure 2.14: LSC elements

Furthermore, It is important for a Universal LSC, to state at which point(s) of the run the LSC should be considered, otherwise the behaviors of the entire system have to be specified in one LSC. The authors in [23] define the *activation condition* and the *pre-chart* of an LSC: The activation condition is a boolean condition, which expresses the activation point of an LSC. The pre-chart allows to specify a prefix or history which must be fulfilled by a run in order to activate the LSC. Pre-charts do not replace the activation condition, but extend it; the activation condition in the presence of a pre-chart indicates the starting point of the prefix. The informal semantics of an LSC with pre-chart is consequently: If the activation condition holds and afterwards

the pre-chart is completed, then the LSC is activated.

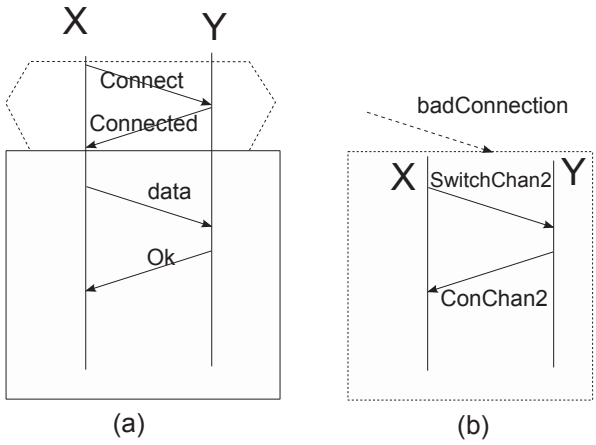


Figure 2.15: Example of LSCs

Figure 2.15 shows an example of two LSCs of a distributed system describing the behaviors of two machines X and Y. The LSC in Figure 2.15-a) is a Universal LSC (solid line) with a pre-chart. When this pre-chart is completed (which means that X sends the message $Connect$ to Y then Y sends the message $Connected$ to X), the chart is activated (X sends the message $data$ to Y then Y sends Ok to X). The LSC in Figure 2.15-b) is an Existential LSC (dashed line) with an activation condition (which is the message $badConnection$). When the activation condition is satisfied this does not necessarily means that the LSC is activated.

2.4.4 Conclusion

MSC is probably one of the most powerful specification models. The main reason is that it allows to give an hierarchical order to the diagrams and, thereby, describe parallel, sequential and alternative scenarios, and to describe non-regular behaviors as well. All this can be presented in a clear and easy way. However, a disadvantage of MSC, which is also common with other high level specification formalisms, is that the descriptions of these scenarios are not precise enough to derive an equivalent code or an abstract machine model, that we will call in the sequel "*implementation*".

2.5 Implementation of Message Sequence Charts

Many researchers consider that, in order to use MSCs in the software life-cycle, it is important that the MSC specification can be translated into distributed state-based specifications (abstract machine models). A natural question is: why not to use directly abstract machine specification? Actually, specifying the system directly with a state-based specification requires explicit identification of states and thus much more consistency when constructing scenarios. This forces the users to reason about their system in terms of states rather than sequences of actions which is very complex specially for large distributed systems.

Then scenario-based inter-object specifications (e.g., via live sequence charts) and state-based intra-object specifications (e.g., via statecharts) are two complementary ways to specify behavioral requirements. This raises the questions of realizability and implementation. The realizability (or implementability) problem is to know whether we can build an abstract machine model with exactly the same behaviors as the given specification. The implementation problem (or the synthesis) consists in building an abstract machine model with exactly the same behaviors as the given specification.

Before formalizing the synthesis problem, we present some of the most important and used abstract machine models.

2.5.1 Abstract machine models

The abstract machine model is the operational model of the system. It describes how each process should behave independently in the system. The code generation is a simple task once an abstract machine model exists. Thus, it is very important to choose a specification model that can be easily translated into an operational model so we can benefit from the specification. As human translation is error-prone, it is important to produce this translation automatically. However, this local view of the specification that each process have in an abstract machine model may add concurrency between the processes. This concurrency might then allow additional behaviors that were not described at the high-level specification. Thus the concurrency is an important point to consider when translating a high-level specification into an abstract machine model.

Several abstract machine models appeared in the literature, next we will present only some of the most popular notations.

2.5.1.1 Petri Nets

Petri nets (PN for short) [68] can be used to describe the state-based behavior of one instance of the system, or the interactions between several instances as well. It was first introduced in the doctoral thesis of C.A. Petri [73]. Since then the PN model has been developed and applied in a wide range of applications like in communication networks, data flow systems, etc [84]. A Petri Net is a directed bipartite graph with two nodes types: The first, called *places*, represent conditions. The second, called *transitions*, represent the events that may occur. These nodes are connected via directed arcs such that these arcs never occur between two nodes of the same type. A PN is defined as follows:

Definition 2.5.1 (Petri net)**.** *A* Petri net *is a tuple* (P, T, F)*, where*

- P *is a finite set of places,*
- T *is a finite set of transitions,*
- $F \subseteq (P \times T) \bigcup (T \times P)$ *is a set of arcs (flow relation).*

Graphically the places are represented by circles and the transitions by dashes. Several works treated the transformation of MSCs into PN as an abstract machine model [20, 72]. A simple way for representing an MSC by a PN is as follows: The head and the end symbols of instances in a bMSC are represented by a start and an end place for each instance. The MSC events are represented by transitions. A token moving through the net represents the control flows within the system. This token moves from start place to the end place passing all along the transitions and places presenting the occurring MSC events. The Figure 2.16 presents the different representations of MSC events in PN: Figure 2.16(a)for the local actions, Figure 2.16(b) for the sent event and Figure 2.16(c) for receive event. Figure 2.17 presents an example of a transformation of a bMSC into a PN based on the elements presented in Figure 2.16: For each instance in the bMSC we get the events that we transform into Petri net fragments. The resulting Petri net fragments are then composed sequentially in correspondence to the bMSC instances. Finally, the Petri net fragments for the instances are composed in parallel (see [47] for more details). In Figure 2.17, markings of places represent these facts about the system:

– The tokens in places p11, p21 and p31 represent respectively the fact that the processes A, B and C have been started. And the tokens in places p13, p24 and p32 represent respectively the fact that the processes A, B and C have ended.

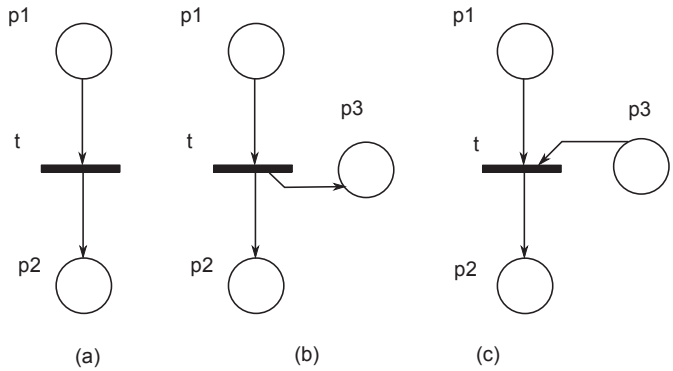


Figure 2.16: The representation of some MSC's events in PN

- p11: represents the fact that the process A has been started, and that A is ready to send the message m1 to the process B.
- p12: A has sent the message $m1$ to B, and that A is ready to send the message m2 to B.
- p13: A has sent the message $m2$ to B, and this is the end place for A.
- p21: represents the fact that the process B has been started, and that it is ready to receive the message m1 from the process A.
- p22: B has received the message $m1$ from A, and that B is ready to send the message m3 to C.
- p23: B has sent the message $m3$ to C, and that B is ready to receive the message m2 from A.
- p24: B has received the message $m2$ from A, and this is the end place for B.
- p31: represents the fact that the process C has been started, and that it is ready to receive the message m3 from the process B.
- p24: C has received the message $m3$ from B, and this is the end place for C.
- A token in the place p121 represents the message m1.
- A token in the place p122 represents the message m2.
- A token in the place p321 represents the message m3.

The transitions represent the following activities in the system:

- t11: A sends the message $m1$ to B,
- t21: B receives the message $m1$ from A,
- t12: A sends the message $m2$ to B,
- t23: B receives the message $m2$ from A,
- t22: B sends the message $m3$ to C,

– t31: C receives the message $m3$ from B,

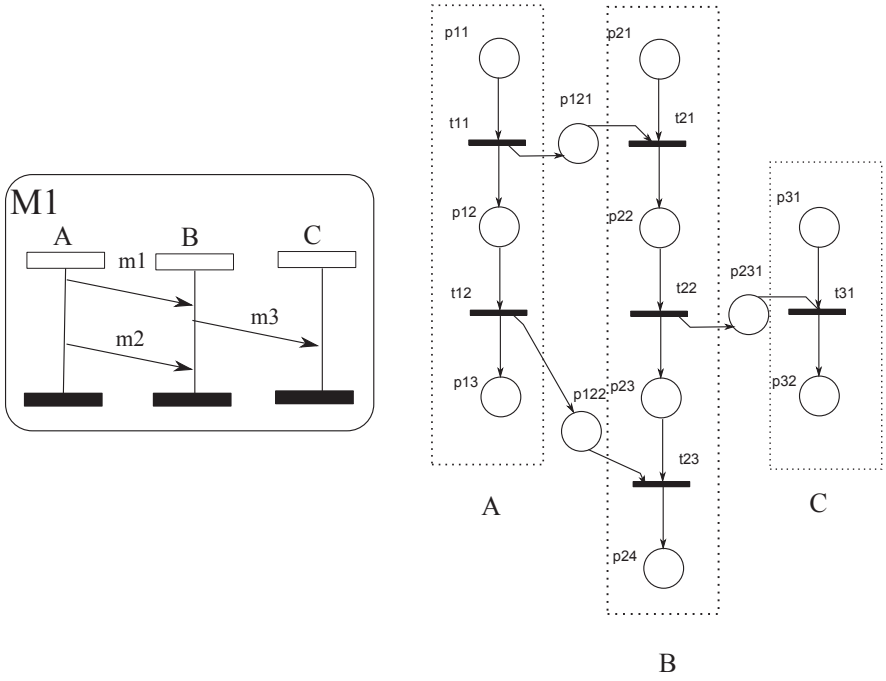


Figure 2.17: The transformation of a bMSC into a PN

The implementation of an HMSC with Petri nets results in additional behaviors [20]. for instance let us consider the example of HMSC of Figure 2.18.
If the process A of the HMSC of Figure 2.18 sends the message $m1$ then the message $m2$ to the process B, then B must receive the message $m1$ before receiving the message $m2$ what might not be the case in the corresponding Petri Net presented in Figure 2.19 Then, one shall notice that HMSCs semantics can enforce messages between a pair of processes to respect FIFO ordering (that we will explain in chapter 3), which cannot 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 [20]. So PN cannot be used as implementation model.

2.5.1.2 Statecharts

Statecharts are synchronous languages originally introduced by Harel in [33]. They are a variant of the Finite State Machines (denoted FSMs). FSM is a model of computation that consists of a set of *states*, a start state, an input alphabet, a transition functions, and accepting states. The computation begins at the start state, then it

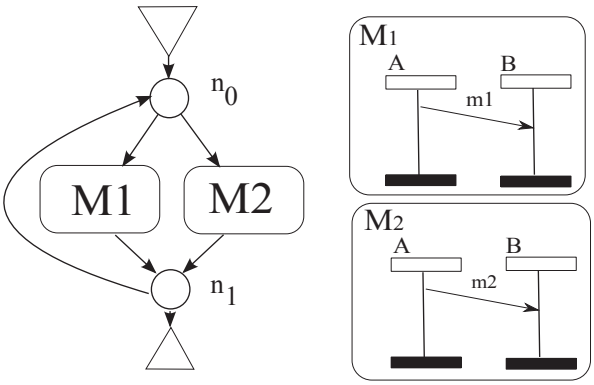


Figure 2.18: Example of HMSC

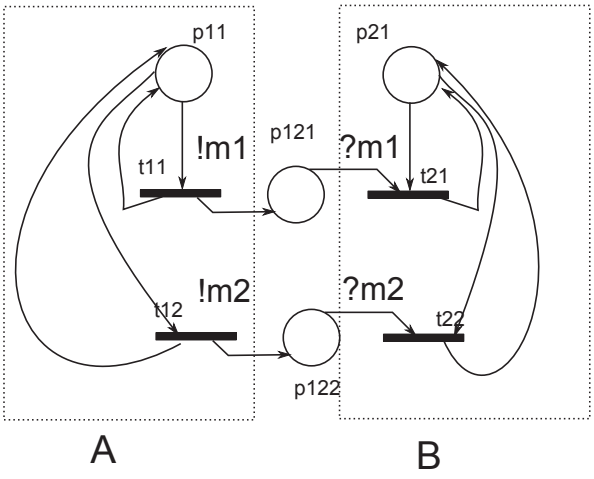


Figure 2.19: The PN implementation of the HMSC of Figure 2.18

changes to a new state for each event/message (an event is somethig that occurs in the system like an input from the environment, a message, etc.) depending on the transition function. Statecharts have extended the FSMs with some additional features like hierarchy and parallelism, and broadcast communications. Statecharts formalism continue evolving over the years, spawning many variants like classical statecharts (or Harel's Statecharts) UML Statecharts, and Rhapsody Statecharts [22].

It is clearly stated in the Z.120 standard [35] that bMSCs and HMSCs depict the behavior of agents that communicate asynchronously, which rules out statecharts as a possible abstract machine model. In [49] the authors consider the transformation of a finite set of bMSCs into statecharts but the communications are supposed synchronous. Some other works deal with the transformation of HMSCs into statecharts but they change HMSCs semantics so that the execution of a bMSC does not start while the execution of the previous bMSC have not yet ended. In the method of synthesis that we propose in chapter 3, we will not change the semantics of the HMSC [58], and yet implement correctly a subset of the language.

2.5.1.3 Communicating Finite State Machine

Communicating Finite State Machines (CFSM for short) [17] appeared as one of the earliest abstract machine models to represent distributed systems [18, 87], and are used for instance in the specification language SDL. A CFSM $\mathcal{A}$ is a network of finite state machines that communicate over unbounded, non-lossy, error-free and FIFO communication channels. One state machine is presented as a directed labeled graph where nodes represent states and edges represent transitions. A transition between two states can be either a *send* or a *receive* or a *local* transition. An edge is labeled by $p!q(m)$ (when the current machine named p sends a message m to another machine named q), or by $p?q(m)$ (when the current machine named p receives a message m from another machine named q), or by a (where a is the name of a local action of the current machine). Each state in a state machine has at least one output edge except the final state. One of the states is identified as its initial state; and all states are reachable from the initial state. A subset of states, called accepting states (or final states), are states that mark a successful run which is a run that ends with emtpy buffers. We give a formal definition of CFSM and their semantics in chapter 3 section 3.
Figure 2.20 presents an HMSC and two CFSMs A and B that describe the behaviors of the two processes in the HMSC. The initial states of these two machines are de-

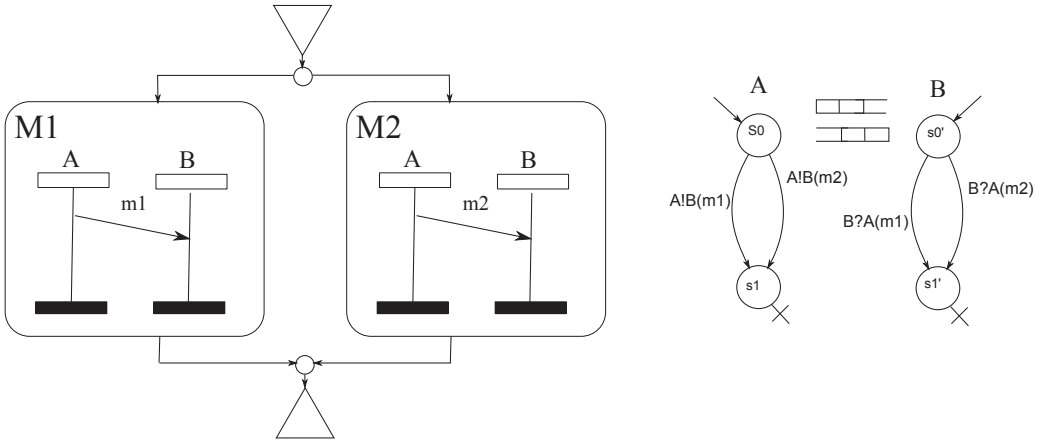


Figure 2.20: Two communicating machines.

noted by a dark incoming arrow and the final states by a cross.

The tight relationship of CFSMs with MSCs is well known [56, 8]. For instance, Lohrey in [56] considers that an accepting run of a CFSM generates in a canonical way an MSC. In the sequel, we choose CFSM model as the implementation model, so we will mainly focus on realizability and implementation problems for HMSCs and CFSMs.

Actually the synthesis of an HMSC into a CFSM might contain deadlocks. For instance let us consider the figure 2.21 that presents an HMSC and its corresponding CFSM. In this example, if the process A sends the message ma to the process B and at the same time the process B sends the message mb to the process A; In this case the two corresponding communicating automata will be respectively at the states $s1$ and $s1'$ with the two messages ma and mb in their respective buffers. This is a deadlock situation.

The semantics of CFSMs is usually defined as the set of runs that do not lead to deadlocks. The events that occur and lead a run to deadlock should not appear in the semantics, thus these events are canceled. We consider that allowing deadlocks in an implementation, and considering that we can simply cancel the events that lead to deadlock is not a realistic solution. In the real life applications (like avionics), events cannot be canceled simply by undoing them. In the synthesis algorithm that we will propose in chapter 3, we consider as semantics of CFSM all prefixes of extensions of the network of machines, including prefixes of executions that end on a deadloack.

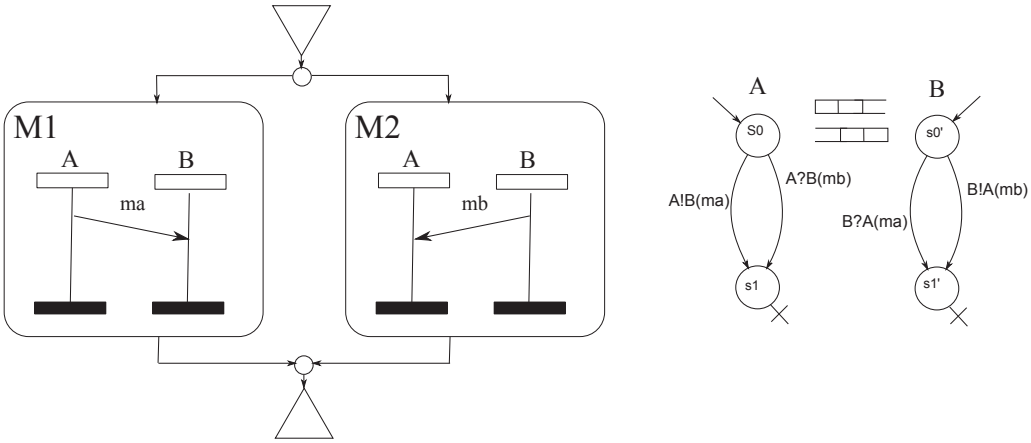


Figure 2.21: Example of a CFSM that might deadlock.

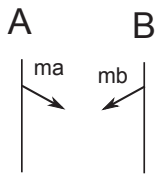


Figure 2.22: A run of the example of Figure 2.21 that deadlocks

2.5.2 The realizability and the synthesis of HMSCs

The synthesis of a scenario-based model consists in building an abstract machine model with exactly the same behaviors as the scenario-based model. Several pathologies in scenario-based models that prevent their synthesis have been studied. An overview of 21 approaches is given in [55] where the authors compare some of the algorithms that generate abstract machine models from scenario-based models that have been proposed in the literature. The differences and similarities of the approaches are identified using two sets of comparison criteria: criteria relevant from a user's perspective (Intended use, Support of parallelism, Support of composition mechanism, etc), and criteria relevant from a technical perspective (Consistency check: The requirements can be semantically inconsistent, Completeness check: the behaviors inferred from the synthesized abstract machine models may not be equal to the behaviors specified by the specification models, etc.). One of their goals is to identify the differences and similarities among approaches and highlight them in the comparison results. The other goal is to explore some of the challenges that current approaches may face are: the implied scenarios (the additional behaviors that were not described in the specification), the consistency (e.g., the synthesized model contains deadlocks), the support of parallelism or concurrency (They noticed that more than half of the approaches do not support parallelism. The reason behind this may be related to the computational complexity typically introduced by the support of parallelism that we explain in chapter 3 section 3.1.4), etc.

Next we will consider works about the synthesis of MSCs specifications. Some of these works consider the synthesis of bMSCs into abstract machine models. A bMSC depicts the exchange of messages among the communicating entities in a distributed system, it contains neither loops nor alternatives and then it corresponds to a single execution of the system that describe a finite set of behaviors. Therefore, a finite set of bMSCs also describes a finite set of behaviors. In [7], the authors study the synthesis of CFSM from a set of bMSCs, and present an algorithm that detects other unspecified and possibly unwanted scenarios called *implied* scenarios. Thereby, if we have no implied scenarios then there is an algorithm that can synthesize CFSM with exactly the same behaviors as the specification. The authors present two notions of realizability, depending on whether the realization is required to be deadlock-free (safe realizability) or not (weak realizability).

However, to provide a more complete description of system behaviors we need to use

richer formalisms and HMSCs have received a quite attention for this. Several works consider synthesis of HMSC specifications into abstract machine models. For instance, [48] considers a synthesis method that translates an HMSC into SDL specifications, by projection (that is build one communicating agent per process) of the HMSC on its instances. However, the generated SDL system allows more traces than those defined by the HMSC specification. This is due to the impossibility of preserving an order between message receptions from different senders. The projection on processes does not preserve this order. It is the same for [20] that considers the implementation of HMSCs by Petri nets but with a larger set of behaviors. In the sequel, and as we have chosen CFSM as the implementation model, we will mainly focus on realizability and implementation problems for HMSCs and CFSMs.

The realizability (or implementability) problem of HMSCs into CFSMs consists in deciding whether we can build a CFSM with exactly the same behaviors as the given HMSC. Some works [8, 83] present two notions of realizability depending on whether we require the implementation to be deadlock-free (safe realizability) or not (weak realizability). The question about the realizability of the HMSCs by CFSMs was studied in several approaches [7, 8, 31]. These studies show that this realizability is in general undecidable, unless the specifications meet some restrictions. In [56], Lohrey prove that the realizability of HMSCs into CFSMs is undecidable for class of general HMSCs. Thus several sub-classes of HMSCs that have synthesis algorithms were presented in the literature. Trivially the realizability of these sub-classes into CFSMs is decidable.

2.5.2.1 Globally-cooperative HMSC

The Globally-cooperative HMSCs have been introduced in [66]. Before introducing the definition of Globally-cooperative HMSCs let us define the communication Graph of a bMSC:

Definition 2.5.2 (communication Graph of a bMSC)**.** *The communication graph of a bMSC M is the directed graph $G(I,\mapsto)$ where I is the set of active instances of M: , and such that for $i \in I$ and $j \in I$, $(i,j) \in \mapsto$, if there exists an event $e{=}i!j(m)$. An MSC is called connected (resp. strongly connected) if its communication graph is connected (resp. strongly connected). Communication graphs are useful to classify high-level MSCs.*

A Globally-cooperative HMSC is defined as follows:

Definition 2.5.3 (Globally-cooperative HMSC)**.** *An HMSC $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ is called globally-cooperative, if for every cycle ρ in H, M_ρ has a weakly connected communication graph.*

In Figure 2.23, the HMSC H_{ngc} is not a globally-cooperative HMSC, since G_{ngc}, the communication graph corresponding to (n_0,M_1,n_0) in H_{ngc}, has two weakly connected components one over A,B and the other over C,D. The HMSC H_{gc} is a globally-cooperative HMSC as G_{gc}, the communication graph corresponding to (n_1,M_2,n_1) in H_{gc}, is a weakly connected graph.

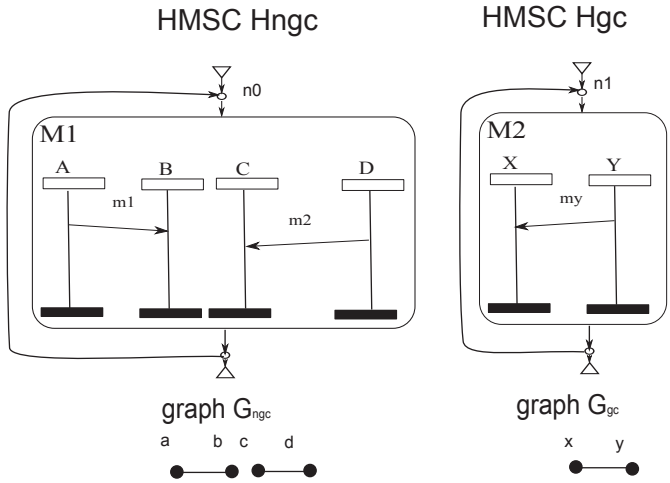


Figure 2.23: globally-cooperative HMSCs

Globally-cooperative HMSCs are always implementable by a CFSM but with possible deadlocks [30]. There is an EXPSPACE-complete algorithm to test whether a globally-cooperative HMSC is implementable with a deadlock-free CFSM and without additional data [56]. However, it is clear that the algorithm is obviously time-consuming, and sometimes even some easily implementable HMSC are considered not deadlock-free implementable, as the globally-cooperative HMSC of Figure 2.24 [29]. At node $n0$ we have a similar situation as for the example of the Figure 2.21 that lead to a deadlock. Such situation will be called a non-local choice, and will be discussed in details in chapter 3.

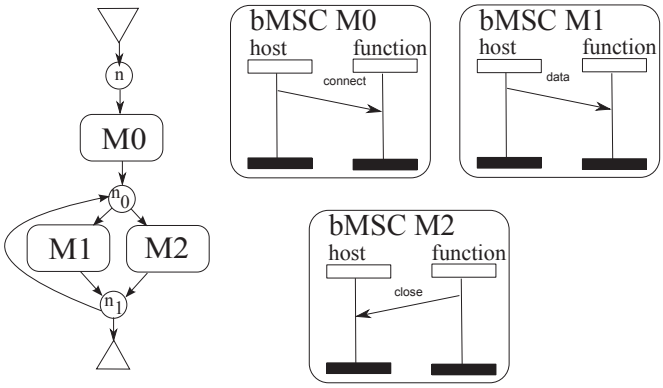


Figure 2.24: HMSC depicting the transactions of usb 1.1

2.5.2.2 Regular HMSC

Another subclass of HMSC, the regular HMSCs, was introduced in [10].

Definition 2.5.4 (Regular HMSC)**.** *An HMSC H is called regular, if every bMSC labeling a loop of H has a strongly connected communication graph.*

The HMSC H_{reg} presented in the figure 2.25 is a regular HMSC. A regular HMSC is a globally-cooperative HMSC with bounded communication channels (buffers have bounded contents in any execution).

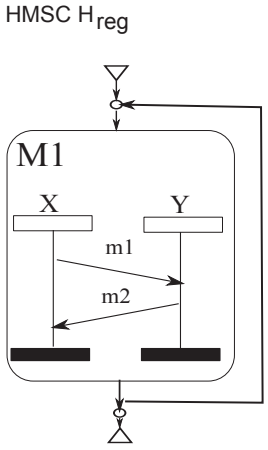


Figure 2.25: Regular HMSC

The authors in [67] proved that any regular set of MSCs admits a deterministic implementation with bounded channel capacities up to some additional message contents

called time-stamps. This result shows the power of adding contents to messages compared to the more restrictive approach proposed in [7] where no additional message content is allowed. For non-FIFO communications systems [66] proved that weak realizability is decidable for bounded HMSCs. The work in [7] was extended in [8] to consider realizability of bounded HMSCs. In [8], the authors proved that for FIFO communication systems weak realizability is, surprisingly, undecidable for bounded MSC-graphs, while safe realizability is in Expspace. However, the question of the exact complexity remains open. In [56], Lohrey prove that for FIFO communications safe realizability is EXPSPACE-complete for bounded HMSCs and that under non-FIFO communication weak realizability is EXPSPACE-hard for bounded HMSCs. [13] extends [67] and consider non-FIFO communication, and identify a subclass of HMSCs (called coherent HMSCs), which are safely realizable with additional message contents. However, checking whether an HMSC is coherent is in general difficult. Coherence is undecidable for HMSCs, and EXPSPACE-complete for locally synchronized HMSCs and for globally cooperative HMSCs. (theorem 5.1 in [13]).

2.5.2.3 Locally-cooperative HMSC

The locally-cooperative HMSCs sub-class was introduced in [31] and is a sub-class of globally-cooperative HMSCs.

Definition 2.5.5 (locally-cooperative HMSC)**.** *An HMSC* $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ *is called locally − cooperative, if for every bMSCs* $M1$ *and* $M2$ *such that (*$n0$,$M1$,$n1$*)* $\in$ $\rightarrow$ *and (*$n1$,$M1$,$n2$*)* $\in$ $\rightarrow$*, the bMSCs* $M1$,$M2$ *and* $M1 \circ M2$ *all have weakly connected communication graphs.*

Figure 2.26 shows an example of a *non-locally cooperative* HMSC H_{nlc}.

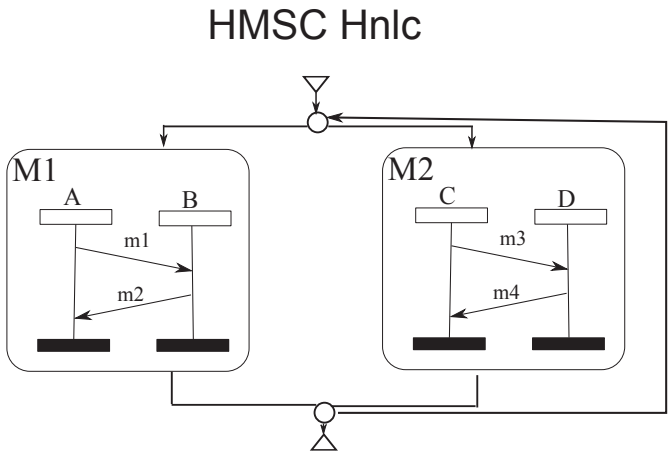


Figure 2.26: Locally cooperative HMSC

In [31], the authors show that locally-cooperative HMSCs can be implemented with an exponential blowup in the number of states and the message contents. Furthermore in general, the implementation is not deadlock-free.

2.5.2.4 Local HMSC

Let us consider the example of Figure 2.24. Node n_1 is a choice node, depicting a choice between two behaviors: either continue to send data (bMSC M_1), or close the data transmission (bMSC M_2). However, at implementation time, this may result in a situation where *host* decides to perform M_1 and $function$ decide concurrently to perform M_2, leading to a deadlock of the protocol. Such situation is called a non-local choice, and causes implementation with deadlocks. It is then safer to implement HMSCs without non-local choices. Ben-Abdallah et al. [15] focus on detecting non-local choices, for which efficient algorithms are given but with restrictions (they only consider nodes not the paths). Intuitively, locality of an HMSC H guarantees that every choice in H is controlled by a unique instance called deciding instance. Checking whether an HMSC is local is decidable [37] The authors in [31] show that local HMSCs can be implemented but with deadlocks and additional message contents, and with initial conditions. The synthesized machines do not deadlock if all the bMSCs in the HMSC have the same set of instances.

We will give more details on local HMSCs in chapter 3.

2.5.2.5 Reconstructible HMSC

Another subclass of local HMSCs that are safely realizable without additional message contents was studied in [37]. In [37], the authors show that the absence of non-local choices is not a sufficient condition to ensure a correct synthesis of CFSM via projection and that a reconstructibility condition is also required (we will present the reconstructible class in more details in chapter 3).

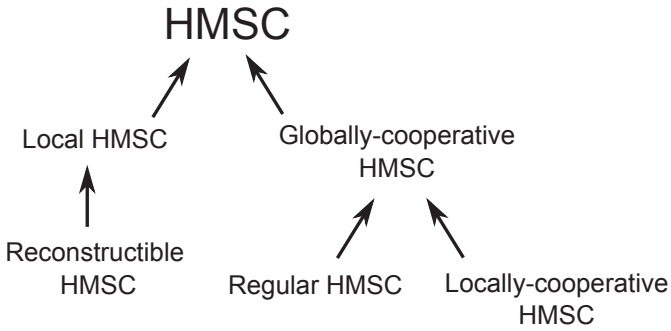


Figure 2.27: Some HMSC sub-classes

Figure 2.27 shows the relations between listed sub-classes. The arrow going from a sub-classe A to a sub-classe B means that A is a sub-classe of B.

2.5.3 Implementation algorithms

Many classical distributed algorithms add data to messages to solve inherent problems of asynchronous systems mainly due to the lack of synchronization between processes. This additional data can be logical clocks as proposed by Lamport [51], or later by Fidge and Mattern [62, 26]. Furthermore, some works like [67, 13, 31] show the importance of adding some finite data to the messages when implementing the specifications. They take advantage of the existing messages to send this additional contents that help to control the computation of the abstract machine in order to achieve the communication sequences as described in the specification. However and as we have already presented, some works [8] and [57] do not allow adding data into messages or adding extra synchronization messages. We think that, just like [67, 13, 31] consider, this is a very restrictive notion of realizability. In the synthesis algorithm that we propose in the next chapter, we allow additional data into messages.

We would like to mention that some 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. In [83], the authors present a technique to detect implied scenarios from a specification consisting of both positive and negative scenarios (positive scenarios are the wanted system behaviors and negative scenarios are the behaviors that the system should not exhibit). The work in [81] 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 can be described as a finite automaton. The work in [54] 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. We refer interested readers to surveys [11, 55] for a more exhaustive list of synthesis approaches with statecharts variants as target language.

Two interesting surveys on synthesis from scenarios have been published [11, 55], where the authors compare and classify many approaches based on the comparison criteria they provided. Liang et al [55] compare the synthesis approaches according to the *source formalism*, the *intended use* (analysis or code generation), the *support for composition operators and parallelism*. The *intended use* of the technique presented in chapter 3 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*. Last, Liang et al check if the synthesis technique checks *correctness* and *completeness* of the synthesized model. Amyot et al [11] use some criteria of [55], and introduce several other criteria 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).

2.6 MSCs tools

MSCs are particularly useful in the early stages of system development procedure. For example, it was reported that an MSC static analysis tool, MINT, helped Motorola reduce appraisal costs and improve productivity [12]. Another software tool, FATCAT [65] has been developed by Motorola UK Research Labs. FATCAT has been used to analyze features developed for 3G handsets, and it has discovered errors in the specifications that had previously gone undetected and which were subsequently discovered only during field testing of pre-release models. Several tools were developed to deal with MSCs, some of them are used for simply display the graphical diagrams like Mscgen, others allow in addition verifications like Möbius or SCStudio (Sequence Chart Studio) tools. Some tools also allow the transformation of existing scenarios into MSC like the PathFinder tool that is used for extracting the core scenarios from existing systems and representing them in MSC, for the maintenance of the system. Other tools allow the transformation of MSC specifications into other formalisms like the tool MSC2SDL and MOST (Moscow Synthesizer tool) [59] that provide a bridge from MSC models to SDL specifications or the SOFAT tool (presented in chapter 5).

2.7 Conclusion

MSCs have proved to be efficient modeling tools to discover errors at early stages of system design and were extensively used (especially bMSCs) to model requirements in distributed systems.

In particular, HMSCs are very expressive, and can model infinite state systems. However, the main difficulty is that general HMSCs are not implementable. It means that system designers cannot benefit from the formal modeling and verification steps performed at early stages of design, to synthesize an implementation and to guarantee its correctness.

In the literature, several works on synthesis use HMSC projection. Most of these works propose solutions for syntactic subclasses of HMSCs only, and usually *local*

HMSCs. Working with local HMSCs is not sufficient to guarantee a correct synthesis. Indeed, the machines synthesized by the MSC2SDL tool [2] or the MOST tool [59] frequently allow for more behaviors than the original specification. To solve this problem, [37] introduced *reconstructible HMSCs* and showed that synthesis by projection is correct for this subclass. The solution in [31] 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 [14] 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 [67] and consider that deadlocked runs are not part of the implemented language. Correctness is an improvement with respect to [2, 59], and completeness an improvement with respect to [31].

In the next chapter, we will define and prove the correctness of a method that allows the implementation of any local HMSC by translating it into an operational model, namely CFSM with controllers and additional messages contents. The translation is done by projecting the HMSC on each active process to get the CFSM. And as it is well known that this solution produces programs with more behaviors than in the specification [37], the proposed model composes the projections with local controllers that intercept the exchanged messages between the automata and tag them with some information to avoid the additional unwanted behaviors. This will prove once again the benefit of using additional data in the contents of messages.

Chapter 3

Local HMSCs : a correctly implementable class of HMSCs

This chapter extends the state of the art by proposing a correct synthesis mechanism for the whole subclass of local HMSCs. The proposed synthesis technique is to project an HMSC on each process participating to the specification. This technique, without additional message contents or control mechanism, 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 [37]. 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.

This chapter is organized as follows: Section 3.1 defines the formal models that will be used in the next sections. Section 3.2 characterizes the syntactic class of local HMSCs. Section 3.3 defines the projection operation, that generates communicating finite state machines from an HMSC, and shows that an HMSC and its projection are not equivalent in general. Section 3.4 proposes a solution based on local control and message tagging to implement properly an HMSC. Section 3.5 compares our approach with existing techniques (also we classify our approach with respect to some criteria for scenario-based synthesis approaches) and finally in this section we conclude and

propose future research directions.

3.1 Definitions

We first define some basic definitions that will be used in this chapter.

3.1.1 Basic definitions around the specification model

For a bMSC M defined as presented in chapter 2, 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.

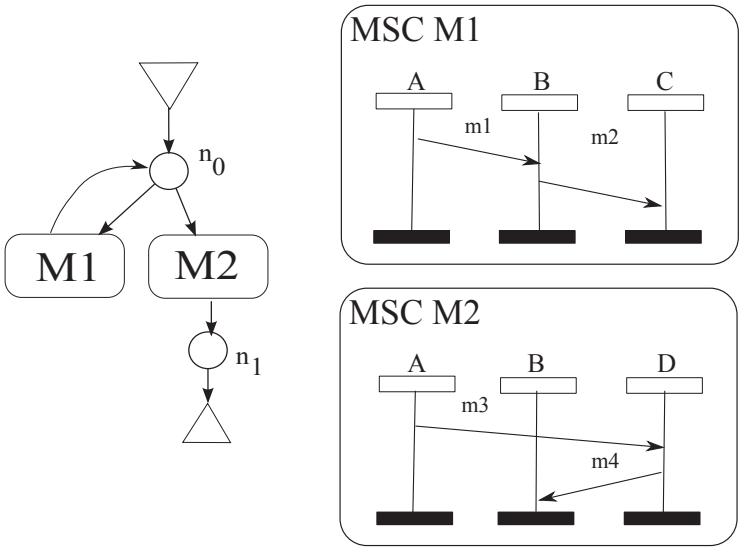


Figure 3.1: An example of local HMSC

We will suppose, without loss of generality, that the HMSCs we have to implement comprise only one hierarchical level, i.e. they are automata whose transitions are labeled by bMSCs. An HMSC can formally be defined as presented in chapter 2, i.e. as a tuple $H = (I, N, \rightarrow, \mathcal{M}, n_0, Fin)$.

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. 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 always be 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.
In the Figure 3.1, we have an HMSC with the set of bMSCs $\mathcal{M} = \{M_1, M_2\}$. The transition relation contains two transitions, namely (n_0, M_1, n_0) and (n_0, M_2, n_1). The behavior M_1 can be repeated an arbitrary number of times, and then be followed by the behavior described in M_2.

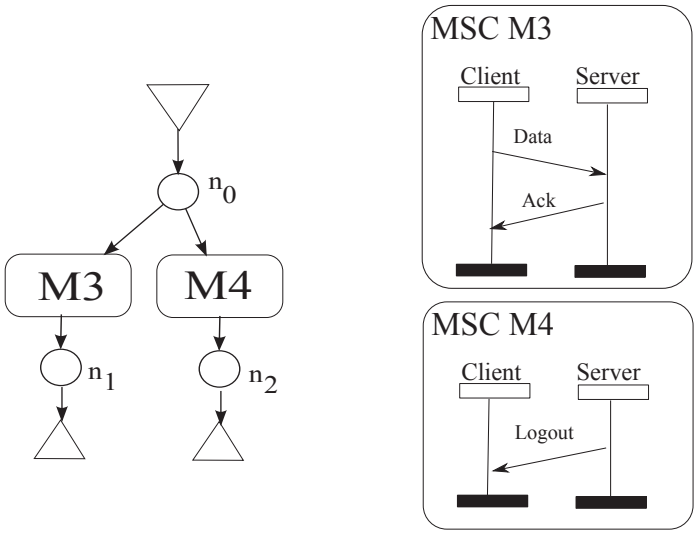


Figure 3.2: a non-local HMSC

3.1.2 Prefix-closed semantics of HMSCs

In chapter 2, we have presented the standard semantics of HMSCs, in terms of generated bMSCs. However, as we target deadlock free implementations, the execution of HMSCs have to take into account that a started and incomplete execution also belongs to the semantics of a system. So in the next sections, executions of bMSCs

will be represented as partially ordered multisets of events (*pomsets*). Furthermore, 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 3.1.1 (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 chapter, 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 3.3 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. 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 2.2.2 and μ is a function that associates the n^{th} sending event from p to q to the n^{th} reception from p on q for every pair of processes $p, q \in I$. Note that this sequencing is not defined if for some p, q, n, the types of the n^{th} sending and reception do not match, that is one event is of the form $p!q(m)$ and the other one $q?p(m')$ 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. We consider that all nodes in an HMSC are accepting nodes and thus we define the prefix *language* of H as the set of behaviors $\mathcal{L}(H) = \bigcup_{\rho \in Paths(H)} Pref(M_\rho)$. To simplify notation, we will write $\rho = n_0 \xrightarrow{M_0} n_1 \xrightarrow{M_1} n_2 \ldots \xrightarrow{M_k} n_{k+1}$ to denote a path $\rho = (n_0, M_0, n_1)(n_1, M_1, n_2) \ldots (n_k, M_k, n_{k+1})$. Note that our definition of the language of an HMSC H includes all prefixes of bMSCs generated by H. A *correct implementation* of an HMSC H is a distributed system reproducing **exactly** (and

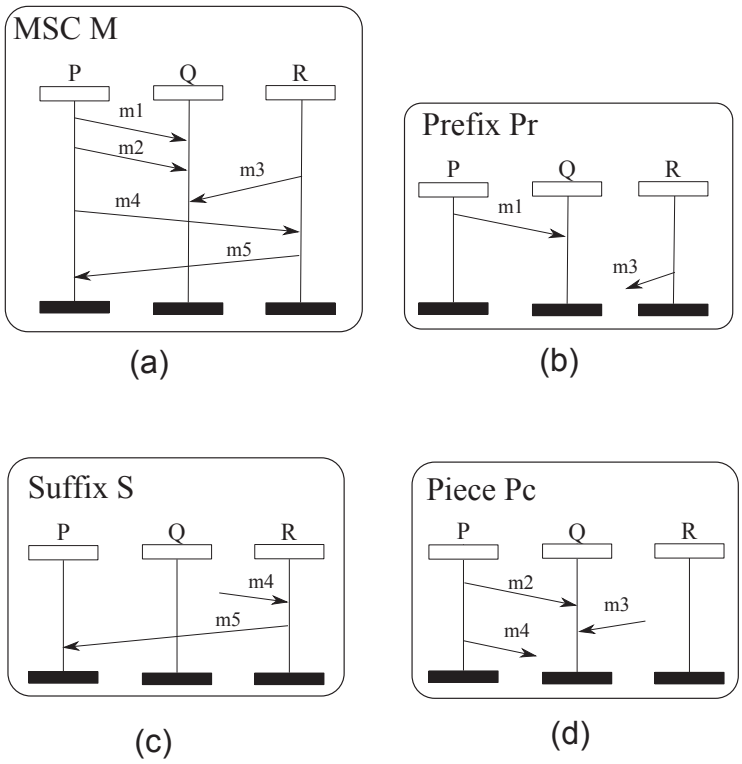


Figure 3.3: A bMSC (a), a prefix (b), a suffix (c) and a piece (d)

nothing more) $\mathcal{L}(H)$.

3.1.3 Semantics of abstract machines

Figure 3.4 describes a CFSM composed of two finite state machines A_{Client} and A_{Server}.

Figure 3.4: Two communicating machines

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. More formally, we can define a communicating automaton as follows:

Definition 3.1.2. *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.*

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 $\mathcal{A} = \parallel_{i \in I} A_i$ 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, and 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 3.1.3. *Let $\mathcal{A} = \parallel_{i \in I} A_i$ 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* $\sigma_1 \dots \sigma_k$.

3.1.4 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 work, 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 [2].

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 chapter 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.

3.2 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 in the bMSC 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 3.2. 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 [82]. 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 [83]. This approach may work for simple cases, but not for all kind of HMSC. 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 [70, 20]). So, one cannot decide if a specification already includes all implied scenarios that appear 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 3.2.1 (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 [15] 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 in which the set of minimal instances in O_{w1} and in $O_{\rho 1}$ is the same, as for all bMSC 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. $\square$

Theorem 3.2.1 (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 also 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 $Min(M) = E \setminus \{f \mid \exists e, e \leq_p f \vee f = \mu(e)\}$. $\square$
From theorem 3.2.1, an algorithm that checks locality of HMSCs is straightforward. It consists in a width first traversal of acyclic paths starting from each choice 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 presented next page describes this procedure. It was originally proposed in [37]. 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 3.2.1, and as we consider a finite set of maximal acyclic paths, termination is guaranteed.

Algorithm 1 LocalChoice(H)

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

3.3 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 $\mathcal{L}(H) = \mathcal{L}(\mathcal{A})$.
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 [41]).

Definition 3.3.1 (Projection)**.** *Let us consider an HMSC $H = (I, N, \rightarrow, \mathcal{M}, n_0)$. The set of events of a bMSC M is denoted by E_M, and the set of events of M located on*

instance i by E_{Mi}. The set E_{Mi} is totally ordered by $\leq_i$. We denote its elements by $e_1, \cdots, e_{|E_{Mi}|}$. 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 $E_i = \bigcup_{M \in \mathcal{M}} E_{Mi}$. We can then define the transition relation $\longrightarrow_i$ as

$$\longrightarrow_i = \{(n, \epsilon, n') \mid \exists (n, M, n') \in \longrightarrow \wedge |E_{M_i}| = 0\} \quad (i)$$

$$\cup \{(n, t(e_1), n') \mid \exists (n, M, n') \in \longrightarrow \wedge |E_{M_i}| = 1\} \quad (ii)$$

$$\cup \{(n, t(e_1), (n, M, n', 1)) \mid (n, M, n') \in \longrightarrow \wedge |E_{M_i}| \geq 2\} \quad (iii)$$

$$\cup \{((n, M, n', k-1), t(e_k), (n, M, n', k)) \mid (n, M, n') \in \longrightarrow \wedge 2 \leq k < |E_{M_i}|\} \quad (iv)$$

$$\cup \{((n, M, n', k-1), t(e_k), n') \mid (n, M, n') \in \longrightarrow \wedge k = |E_{M_i}|\} \quad (v)$$

In the previous definition(i) corresponds to cases when the instance is not concerned by the bMSC M, (ii) is for when a single event e_1 occurs on the instance i in M, $(iii), (iv)$ and (v) correspond to when the set of events occurring on the instance i when running M is at least two events: (iii) corresponds to the transition after the execution of the first event, (v) corresponds to the transition after the execution of the last event, and (iv) corresponds to the transitions after the execution of the intermediate events.

The synthesis by projection from the HMSC of Figure 3.1 produces the CFSM of Figure 3.5. 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 3.2 produces the CFSM of Figure 3.4. 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 3.2, so the CFSM of Figure 3.4 cannot be considered as a correct implementation. In general, the projection of an HMSC on its instances can define more behaviors than

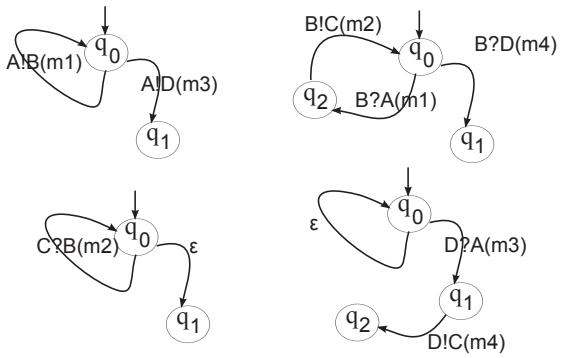


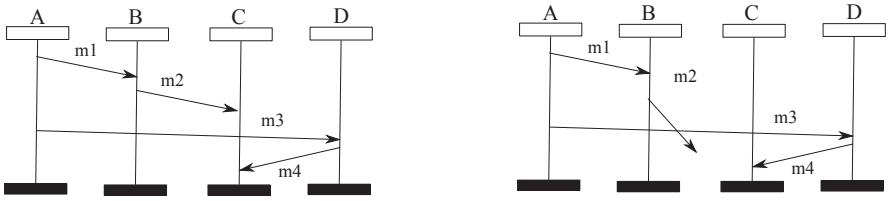
Figure 3.5: The instance automata projected from the HMSC of Figure 3.1

the original specification, but can also deadlock (as the run that we have presented in chapter 2 Figure 2.22). Hence, synthesis by projection on instance is not correct for any kind of HMSC. It was proved in [37] that the synthesized language contains all runs of the HMSC specification.

Theorem 3.3.1 ([37])**.** *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 chapter, we will only consider local HMSCs. However, we can show that this locality is not sufficient to ensure correctness of synthesis. Let us consider the projection of H in Figure 3.1 on all its instances given in Figure 3.5. A correct behavior of H is shown in Figure 3.6-a), while a possible but incorrect behavior of the synthesized automata is shown in Figure 3.6-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, C does not have enough information to decide to delay the consumption of m_4, and hence exhibits an unspecified behavior.
As we consider prefix closed semantics, i.e. we disallow CFSM to deadlock, this behavior is part of the language of the synthesized CFSM.
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 3.3.2. *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*



(a) (b)

Figure 3.6: *a*) A correct behavior of $M_1 \circ M_2$ of Fig. 3.1 a possible distortion due to the loss of information on projected instances.

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 [37]. This work 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.

Before showing how to decide the reconstructibility of an HMSC, let us first give the definition of the message-transitive closure. The message-transitive closure is defined as follows:

Definition 3.3.3. *The message-transitive closure (or mt − closure, for short) of a partial order relation R is written* R^{*mt}*, and is a relation* R' *such that (e,e') ∈* R' *if and only if:*

- *i)* $(e, e') \in R$*, or*
- *ii)* $\exists e" \in E$ *such that* $eR'e" \wedge e"R'e'$*, or*
- *iii)* $\exists e_1, e_2 \in E^2$ *such that* $\phi(e_1) = \phi(e_2) \wedge e_1 R' e_2 \wedge \mu(e_1) = e \wedge \mu(e_2) = e' \wedge \phi(e) = \phi(e') \wedge (e', e) \notin R$*. (*$\mu(e_x) = e_y$ *means that* e_x *is a message emission and* e_y *is the corresponding reception).*

For instance, in Figure 3.7 e_1 and e_2 are two messages sending occurring on the same instance and e and e' are their respective corresponding messages receptions on another intance. The two events e_1 and e_2 are ordered as they occur on the same instance then based on the rule (i) of the $mt-closure$ definition, (e_1, e_2) is an element of R^{*mt}. Figure 3.7 also illustrates an example on rule (ii) that considers the transitivity on the causality relation between events. The events e_1 and e_2 ordered, and the events e_2 and e' (a sending and receiving of the same message) are ordered, then e_1 and e' are ordered. Finally, based on the rule (iii) the events e and e' are ordered and the order between message receptions is the same as the order between the corresponding emissions (e precedes e').
The message-transitive closure R^{*mt} (or $mt-closure$, for short) is a closure operation then any element $e_1, e_2 \in E^2$, $(e, e') \in R$,(R is a partial order relation) must be in R^{*mt} (in Figure 3.7 e_1 and e_2 are ordered as they are message emissions occurring on the same instance, then (e_1, e_2) is also an element of R^{*mt}). R^{*mt} is a transitive on the causality relation between events (in figure 3.7 we have the e_1 and e_2 ordered, and the events e_2 and e' are ordered then e_1 and e' are ordered). Finally, for e_1 and e_2 were message emissions, and e and e' are the corresponding reception . As no ordering between e and e' exist, and as any pair of events of the same instance must be ordered, the order between message receptions is the same as the order between the corresponding emissions (in Figure 3.7 e and e' are then ordered). The formal definition of R^{*mt} is given in the appendix.

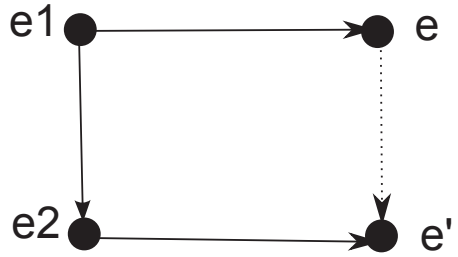


Figure 3.7: Order on events

As for local choices, reconstructibility property can be decided on cycles and paths originating from a choice.

Proposition 2. $\forall$ *c, choice of H, c is reconstructible if for any pair of branches* B_i, B_j *such that* $\exists p$, *path from c and* $O_p = B_i \circ B_j$, *for any non-deciding instance x,* $(min_x(B_i), min_x(B_j))$ *is reconstructible from* $\leq_{B_i \circ B_j}$ - $(E_{i_{|x}} \times E_{j_{|x}})$ *by mt-closure.*

The algorithm 2 is to decide the reconstructibility of an HMSC:

Algorithm 2 Reconstructible(H)

for c choice node of H **do**
  P = $\{(c.n, M) \mid c \xrightarrow{M} n\}$
  $C = \emptyset$ /* Cycles */
  MAP = $\emptyset$ ***/*Maximal acyclic paths*/***
  while $P \neq \emptyset$ **do**
    $C = C \cup \{M \circ M' \mid (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} c\}$
    $MAP = MAP \cup \{M \circ M' \mid (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} n \wedge n \in w\}$
    $P = \{(w.n, M \circ M') \mid (w = c.n_1..n_k, M) \in P \wedge n_k \xrightarrow{M'} n \wedge n \notin w\}$
  end while
  for $(B_i, B_j) \in C \times (MAP \cup C)$ **do**
    if $\exists x \in I \mid (min_x(B_i), min_x(B_j)) \notin (B_i \circ B_j \text{ - } (B_{i_{|x}} \times B_{j_{|x}}))^{*mt}$ **then**
      Order cannot be reconstructed
    end if
  end for
end for

Let us consider the example of Figure 3.1, 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 its instances, as in Figure 3.5. 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 ($L = q_{1,A}.q_{0,B}.q_{0,C}.q_{2,D}, W = \{\epsilon, \ldots w_{BC} = m2, w_{DC} = m4, \ldots \epsilon\}$). 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 3.3.2 ([37])**.** *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 3.3.2, the communicating automata synthe-

sized by projection 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 3.3. If the answer is no, then synthesize the CFSM with their controllers, as proposed in section 3.4.

3.4 Implementing HMSCs with message controllers

The class of reconstructible HMSCs shown in section 3.3 is contained in the class of local HMSCs. This subclass is quite restrictive (for instance, the HMSC of Figure 3.1) is not reconstructible, and hence cannot 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. Yet, the correct behavior still exists in the synthesized machines, as proved by theorem 3.3.1. Hence, a major objective to achieve correct synthesis is to prevent unspecified behaviors.

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 [62, 26], 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 section 3.3 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 $((n_1, M_1, k, n_2), p!q(m), (n_3, M_2, k', n_4))$ (respectively $((n_1, M_1, k, n_2), p?q(m),$ $(n_3, M_2, k', n_4))$) in A_i, by a transition of the form $((n_1, M_1, k, n_2), p!C_p(q, m, b),$ $(n_3, M_2, k', n_4))$ (respectively $((n_1, M_1, k, n_2), p?C_p(q, m, b), (n_3, M_2, k', n_4))$), 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). Note that this possibility of reading buffers contents without consumption slightly extends the expressive power of CFSM, without changing their mere automata. 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).

3.4.1 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 3.8 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.

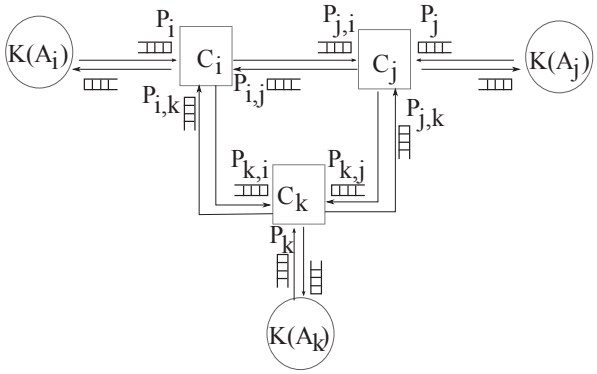


Figure 3.8: The distributed controlled architecture.

3.4.2 Tagging mechanism

Vector clocks are a standard mechanism to record faithfully executions of distributed systems (see for instance [25, 61]), or to enforce some ordering on communication events [74]. 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.
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 3.2, 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 3.4.1 (Choice clocks)**.** *A* choice clock *of an HMSC H is a vector of $\mathbb{N}^{\mathcal{B}_H}$. Let $\rho = n_0 \xrightarrow{M_1} n_1 \xrightarrow{M_2} n_2 \ldots \xrightarrow{M_k} n_k$ be a path of H. The* choice clocks labeling *of O_ρ is a mapping $\tau : E_{O_\rho} \longrightarrow \mathbb{N}^{\mathcal{B}_H}$ such that for every $i \in 1..k, e \in M_i$, $\tau(e)[b]$ is the number of occurrences of branch b in $M_1 \circ \cdots \circ M_i$.*

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

Proposition 3. *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 3.4.2 (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 3.1, the choice clock is an integer vector indexed by b_1, b_2, where $b_1 = (n_0, M_1, n_0)$ and $b_2 = (n_0, M_2, n_1)$. In M_1 and M_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 the process i.

In the rest of the chapter, 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 3). 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 (see Figure 3.9) 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 (see Figure 3.10) 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 $P_{i,j}$ cannot 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.

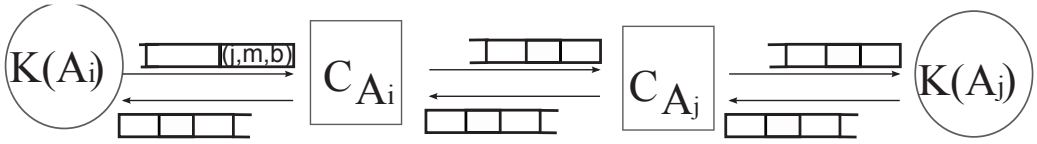


Figure 3.9: The first case showing a state of the buffer of the controller C_i

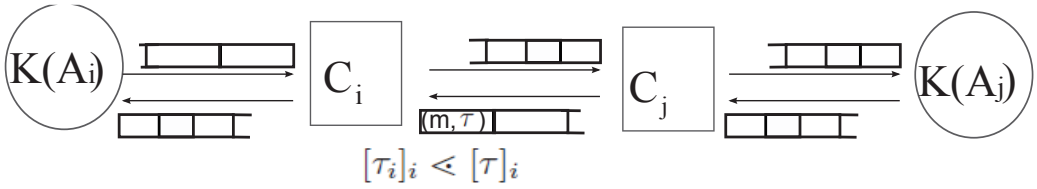


Figure 3.10: The second case showing a state of the buffer of the controller C_i

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

Algorithm 3 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)
/* (j, m, b) ***is the first message of a new branch*/***
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 cannot be consumed because it does not have the right sequence number */
end if

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 $\parallel_{i \in I} (K(A_i)|C_i)$ 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 3.8, 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 [43]. Hence, our controlled automata could be easily encoded as an SDL specification. Last, note that adding controllers to our synthesis architecture does not really increase the expressive power of the network of machines, as CFSMs can already simulate Turing machines. Considered individually, processes 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 3.3 have to play exactly the sequences of events described by the original HMSC to be a correct implementation. Note that as CFSM are Turing powerful, one could simulate the behavior of each controller with a CFSM. However, this would result in a less concise and less intuitive model.

3.4.3 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 cannot require that $\mathcal{L}(H) = \mathcal{L}(\parallel_{i \in I} (K(A_i)|Ci))$ 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 its controller as a communication of a message m from p to q.

Definition 3.4.3. *Let $O = (E, \leq, t, \phi, \mu)$ be a prefix in $\mathcal{L}(\parallel_{i \in I} (K(A_i)|Ci))$. 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 i^{th} sending from p to q with the i^{th} reception on q from p for every pair of processes.*

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

Theorem 3.4.1. *Let H be an HMSC, and let $\parallel_{i \in I} K(A_i)|C_i$ be its controlled synthesis. Then, $Ru(Unc(\mathcal{L}(\parallel_{i \in I} K(A_i)|C_i))) = \mathcal{L}(H)$.*

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 appendix 7.1.

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.

3.5 Conclusion and future work

We have proposed a synthesis framework that produces an implementation for local choice HMSCs into a CFSM model. This synthesis works with additional processes that tag messages and delay them to ensure correct ordering of message receptions.

Actually, the proposed synthesis technique is to project an HMSC on each process participating to the specification. This technique is correct for a sub-class 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 [37]. When an HMSC is not reconstructible, we compose the projections with controllers, that intercept messages between processes. For each process there is a controller that tags the outgoing messages with sufficient information about ordering of messages coded as vectors, or delay some messages receptions according to their tag. This avoids the additional behaviors that appear in the sole projection. The derived CFSMs are correct and complete by construction, i.e. they exhibit exactly the same behaviors as the original description.

Based on the criteria presented by Liang et al [55] and compared to the presented approaches,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 3.1.4: parallel frames introduce non-determinism leading to incorrect synthesized behaviors, and may force implementations to have an unbounded number of control states. The synthesis proposed in this chapter derives local finite state machines, which are controlled asynchronously by machines able to delay some messages. As for the criteria of whether the synthesis technique checks *correctness* and *completeness* of the synthesized model: Our synthesis approach is not concerned by these criteria, as the derived CFSMs are correct and complete by construction. On the other side, based on the criteria presented by Amyot et al [11] and compared to the presented approaches in [11], 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, are not addressed 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 [16], but the proposed model must be highly non-deterministic in order to implement dynamic MSCs.

We note that to keep the construction of CFSM simple, we have supposed FIFO semantics of communications, and we will hence suppose that the HMSCs that we implement do not contain message overtaking. However, the extension of our implementation to models that allow message overtaking should be easy. One fact worth mentioning again is that the controllers are purely asynchronous, which leaves a lot of freedom to choose a particular architecture. In a real implementation, one may suppose that a process and its controller are implemented on the same machine, but this is not mandatory. The use of controllers allows us to make only minor changes to the CFSMs (tagging outgoing messages). Besides, 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.

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 the tool SOFAT [36], 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 (see chapter 5).

In terms of future work concerning the synthesis algorithm, there exist various opportunities for extending our work:

- The integration of data is challenging issue. The techniques proposed in this the-

sis only address the control flow as 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 [4], 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 as correctness criterion.

- 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 cannot 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.

- Another possible perspective can be to study whether asynchronous controllers can in addition enforce properties such as boundedness of buffers, avoidance of a given configuration, etc.

Chapter 4

Localization of HMSCs

As already shown, in the previous chapters, HMSCs are not correctly implementable in general, and the question of whether an HMSC specification can be implemented by communicating machines is undecidable in general [56, 7]. However, several subclasses of HMSCs can be implemented using a simple projection operation and controllers, such as local HMSCs for which we have presented a correct implementation technique in chapter 3.

In this chapter we propose a new technique to transform an arbitrary HMSC specification into a local HMSC, hence allowing a correct implementation. In other terms, we propose to extend the possibility of automated production of CFSMs by the use of a **localization procedure** that transforms any non-local HMSC into a local one. It guarantees that every choice in the transformed local HMSC has a *leader process*, which chooses one scenario and communicates its choices to the other processes. This can be achieved by adding new messages and processes in scenarios.

Trivial but uninteresting solutions to the localization problem exist, like extending bMSCs of the HMSC in such a way that: They all contain a fixed process, designated as a leader for all choices, and messages from this process to all other instances preceding any event in the bMSC. This solution is trivial but it may cause many changes to the specification namely the set of messages and processes in the bMSCs. We are thus interested in finding solutions with the minimal changes to the specification. For instance we can search for the solution that adds the minimal number of messages to the original specification. We propose to address the localization problem with

a constraint optimization technique. We build a constraint model where variables represent leader processes and processes contributing to a scenario and constraints ensure that an HMSC is local. A cost function is then proposed to evaluate the cost of a solution (This cost function must then be minimized by solutions). for instance the number of added messages.
This chapter is organized as follows: Section 4.1 gives an example on how to make an HMSC a local one. Section 4.2 defines localization of HMSCs. Section 4.4 proposes an encoding of minimal localization as a constraint optimization problem, and shows the correctness of the approach. Section 4.5 describes an experimentation conducted to evaluate the performance of our localization procedure, and comments the results. Section 4.6 concludes this work.

4.1 Example

Let us consider the example of Figure 4.1, the HMSC H_{nl} is a non-local HMSC. Replacing M_1 by the bMSC M_3 of Figure 4.2 solves the non-local choice problem. Similarly, replacing M_1 and M_2 respectively by M_4 and M_5 solves the the non-locality problem, but needs more messages.

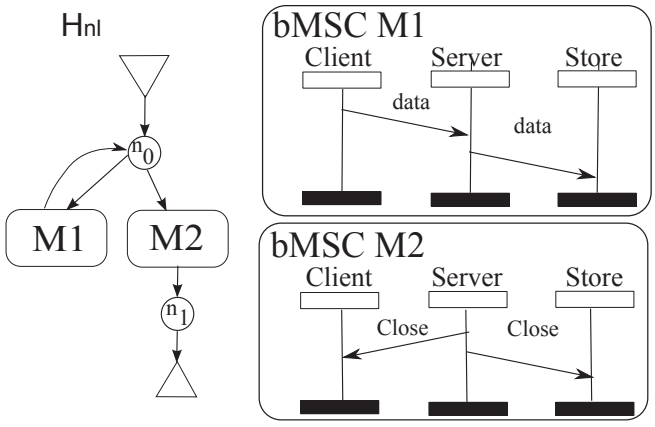


Figure 4.1: Example of a non-local HMSC

This example raises several remarks. First, the proposed transformations are purely syntactic, and modifying the set of minimal instances does not always produce a meaningful specification. For this reason, the examples exhibit changes involving a single message type m. A meaning for additional message has to be chosen adequately

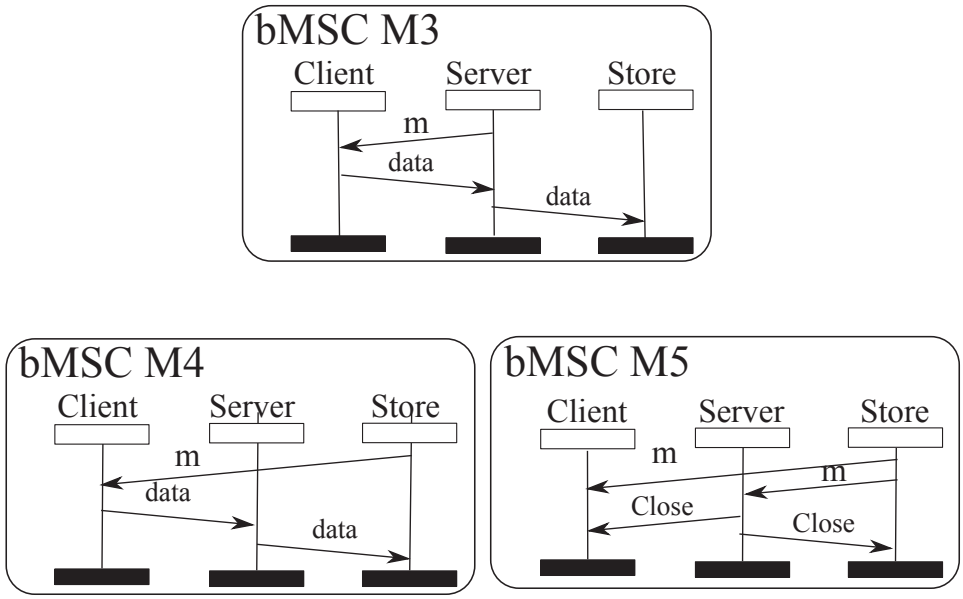


Figure 4.2: Solutions for localization of HMSC in Figure 4.1

by the designer once an HMSC is localized. The second remark is that there are several possibilities for localization. The first solution proposed adds one message in bMSC M_1 to obtain M_3. The second solution adds one message to M_1 and two messages to M_2, and one can notice that in M_5, the message between *Store* and *Client* is useless. Indeed, there exists an infinite number of transformations to localize an HMSC. This calls for the following solutions: We want to restrict to cheapest solutions (for instance solutions with a minimal number of added messages). As we will show later, once a deciding instance for a choice is fixed, one can compute the minimal number of messages needed to localize this choice. As a consequence, the solutions to a localization problem can be given in terms of choosing a deciding instances at each choice, and instances participating to bMSCs. Then, the localization can be easily tuned using different cost functions.

4.2 Localization of HMSCs

In this section, we show how to transform a non-local HMSC into a local one. This procedure called *localization* consists in choosing a single deciding instance for each bMSC M in the HMSC so that all choices become local, and then ensuring that all other instances execute their minimal events only after the first event (the *choice*) of

the deciding instance. This is done by adding messages, as in the examples of Fig. 4.2.

Definition 4.2.1. *Let M be a bMSC over a set of events E and processes P, with minimal events $e_1, \ldots, e_k$. A* localized extension *of M is a bMSC M' over a set of events $E' \supseteq E$ and over $P' \supseteq P$, such that there exists a minimal event $e_{min} \in E'$ (that is $e_{min} \leq' f$ for every $f \in E'$) and for every $e \leq f \in E$, we have $e \leq' f$. The unique minimal instance in a localized bMSC M is called the* leader *of M.*

Note that as there exists an infinite number of extensions for a bMSC M, choosing extensions that are as close as possible to the original model is desirable. The impact of localization can be simply measured as the number of added messages. A more generic approach is to associate a cost to communications between processes, and to choose extensions with minimal cost. This makes sense, as for instance the cost and delays for communications via satellite are higher than with ground networks. Similarly, the configuration of a system may prevent two processes p and q from exchanging messages. To avoid solutions with communications between p and q, one can design a cost function that associates a redhibitory (or even infinite) cost to such forbidden communications.

For a given bMSC M with k minimal events, there exists a localized extension over the same set of processes that contains exactly $k-1$ additional messages. This localized extension is built when one picks up a deciding instance d among the minimal instances of M, and create causal dependencies from the minimal event on instance d to all other minimal events with additional messages. Only $k-1$ messages are necessary in this case, regardless of their respective ordering and place of insertion in the original bMSC. The figure 4.3 presents two propositions Ma and Mb of localization for the bMSC $M1$, both with $k-1$ additional messages. Another possibility is to pick up another non-minimal process among those of M that do not carry a minimal event as a leader, or even add a new process to M. In such cases, a localized extension can always be built with exactly k additional messages. The figure 4.3 presents Mc as a proposition of localization for the bMSC $M1$, with k additional messages.

Localization of HMSCs is more complex than localization of bMSCs. For each non-local choice c, we have to ensure that *every branch* leaving c has the same leader. Hence, this is not a property purely local to bMSCs. As for bMSCs, we can define a notion of localized extension of a HMSC as follows:

Definition 4.2.2. *Let $H = (I, N, \longrightarrow, \mathcal{M}, n_0)$ be an HMSC. $H' = (I, N, \longrightarrow', \mathcal{M}', n_0)$ is a* localized extension *of H iff*

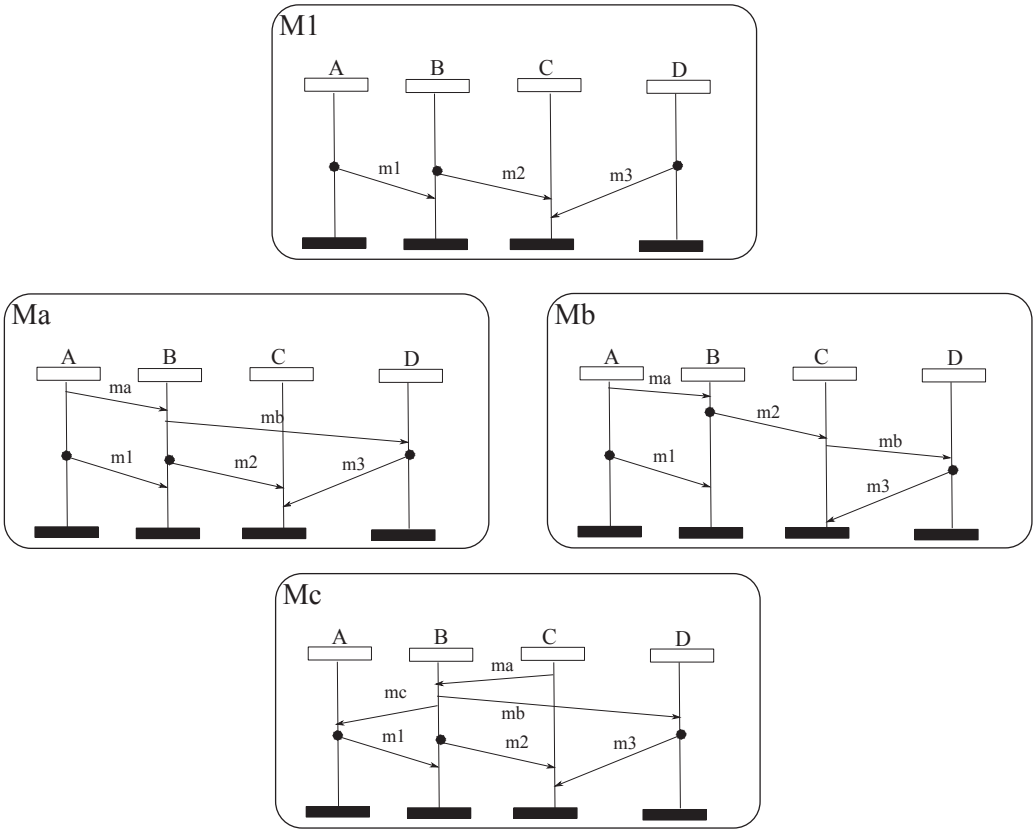


Figure 4.3: localization of a bMSC

- *there is a bijection* $f : \mathcal{M} \rightarrow \mathcal{M}'$ *such that* $\forall M \in \mathcal{M}$, $f(M)$ *is a localized extension of* M,
- $\longrightarrow' = f(\longrightarrow)$,
- *and* H' *is a local HMSC.*

Localizing an HMSC H consists in finding $\mathcal{M}'$ and the bijection f. As mentioned above, as there exists a (potentially) infinite number of solutions, we consider the solutions with the smallest number of changes to the original model. We propose to address this problem with a cost function $\mathcal{F}$ that evaluates the cost of each possible transformation of H. The goal of our localization algorithm is thus to minimize $\mathcal{F}$.

The cost function $\mathcal{F}$ is defined based on the criteria that we want to consider. For instance we can affect a cost to each communication channel based on economic criteria (for instance because of the type of the communications: satellite, ethernet, or others), security and safety criteria (for instance some channels are safer than others, etc.), time criteria (the delay induced by each channel might be different from the others). For the sake of simplicity in the rest of the chapter, $\mathcal{F}$ counts the total number of messages and instances added in $\mathcal{M}'$.

4.3 Messages and processes counting cost function

The cost function $\mathcal{F}$ that we will consider counts the total number of messages and instances added in $\mathcal{M}'$. In this case, $\mathcal{F}$ is defined as a sum of individual costs of modifications. Formally,

$$\mathcal{F}(H, H') \triangleq \sum_{M \in \mathcal{M}} c_{M,f(M)}$$

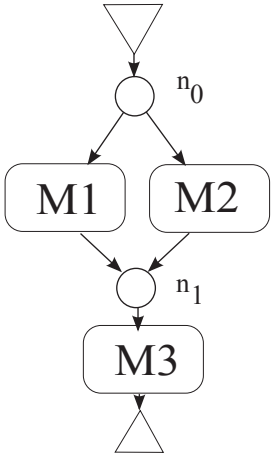
where $c_{M,M'}$ is the individual cost to transform M into M'. When H is clear from the context, we will write $\mathcal{F}(H')$ instead of $\mathcal{F}(H, H')$. Let $M_i \in \mathcal{M}$ be a bMSC, $M'_i = f(M_i)$, I_{M_i}, $I_{M'_i}$ be the set of instances in M_i and M'_i. Let $k = |min(M_i)|$ be the number of minimal instances in M_i, l be the leader instance of M'_i, and $x = |I_{M'_i}| - |I_{M_i}|$ be the number of new instances in M'_i. We choose the constants $\theta_1 \in [0, 1]$ and $\theta_2 \in [0, 1]$ and define the cost c_{M_i,M'_i} for transforming M_i into M'_i as follows:

$$c_{M_i,M'_i} \triangleq \begin{cases} x * \theta_1 + (k + x - 1) * \theta_2 & \text{if } l \in \phi(min(M_i)) \text{ or } l \notin I_{M_i} \\ x * \theta_1 + (k + x) * \theta_2 & \text{otherwise} \end{cases}$$

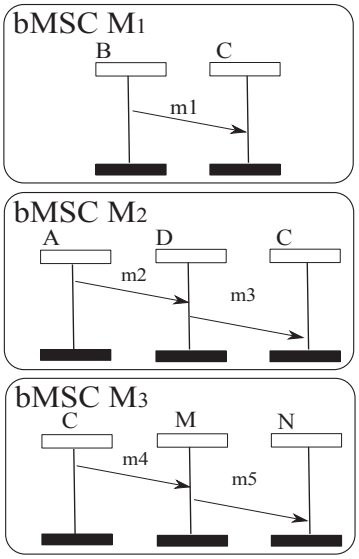
We already know that the number of messages to add is at most $k-1$ if we have k minimal instances. Adding x instances to M_i hence yields adding $(k+x-1)$ messages if l is chosen among the minimal instances of M_i or among the new instances. Similarly, if the leader instance is chosen among instances that are not minimal w.r.t the causal ordering, one needs to add $k+x$ messages to localize M_i. The values θ_1 and θ_2 are chosen to penalize more the number of added processes or the number of added messages.

Let us illustrate the computation of $\mathcal{F}$ on an example. Let H_c be the HMSC of Figure 4.4-a) with the bMSCs of Figure 4.4-b), and let H'_c (presented in Figure 4.4-a) with the bMSCs of Figure 4.4-c)) be a localization of H_c. As mentioned above, $\mathcal{F}(H_c, H'_c) = c_{M_1,M'_1} + c_{M_2,M'_2} + c_{M_3,M'_3}$. The leader of M'_1 is A and, as A is not an instance of M_1, then $c_{M_1,M'_1} = \theta_1 + \theta_2$ (there is a single message and an instance added in M'_1). There is no changes in M'_2 and M'_3 compared to M_2 and M_3 then $c_{M_2,M'_2} = 0$ and $c_{M_3,M'_3} = 0$. As a result, $\mathcal{F}(H_c, H'_c) = \theta_1 + \theta_2$. One can easily notice that H'_c is local. If we compare H_c with another localization H''_c, presented in Figure 4.4-a) with the bMSCs of Figure 4.4-d), we get that $c_{M_1,M''_1} = \theta_2$, $c_{M_2,M''_2} = \theta_2$, $c_{M_3,M''_3} = 0$ and finally, $\mathcal{F}(H_c, H''_c) = 2 * \theta_2$. The values of θ_1 and θ_2 decide which one of two proposed localization is better. If $\theta_1 = 0.5$ and $\theta_2 = 1$ then $\mathcal{F}(H_c, H'_c) < \mathcal{F}(H_c, H''_c)$ and thus, localization H'_c shoud be preferred to H''_c. On the other side if $\theta_1 = 1$ and $\theta_2 = 0.5$ then $\mathcal{F}(H_c, H''_c) < \mathcal{F}(H_c, H'_c)$ and thus, localization H''_c shoud be preferred to H'_c. This example shows that the cost function influences the choice of a particular localization solution.
The graphs of Figure 4.5 shows how the values of θ_1 and θ_2 affects which localization H'_c or H''_c is better (θ_1 is used a x coordinates, θ_2 as y coordinates, and z axis as the cost). For a pair of values (x,y), the lowest function is the best.
The cost function $\mathcal{F}$ defined above that counts the number of new messages and processes in bMSCs is only an example, and other functions can be considered. For instance, a cost function can consider concurrency among events as an important property to preserve, and thus impose a penality everytime a pair of events e, e' is causally ordered in $f(M)$, but not in M.

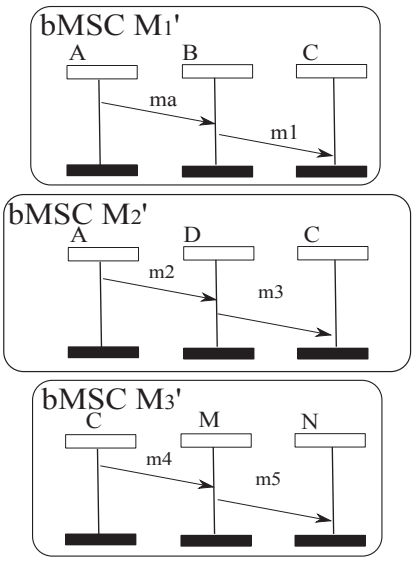
Note also that several localization solutions can have the same cost. For instance, if



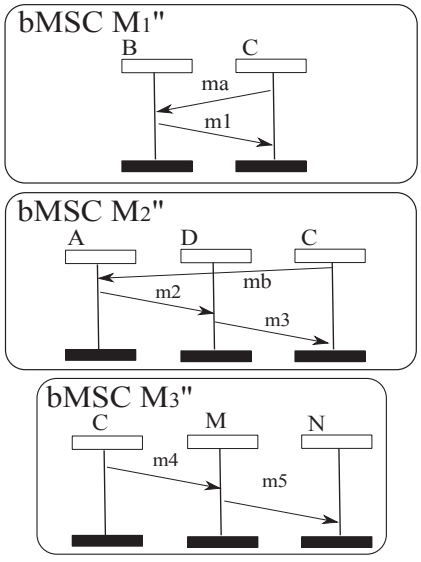
(a) structure of HMSCs Hc , Hc' and Hc"



(b) bMSCs for HMSC Hc



(c) bMSCs for HMSC Hc'



(d) bMSCs for HMSC Hc"

Figure 4.4: Localizing the HMSC Hc

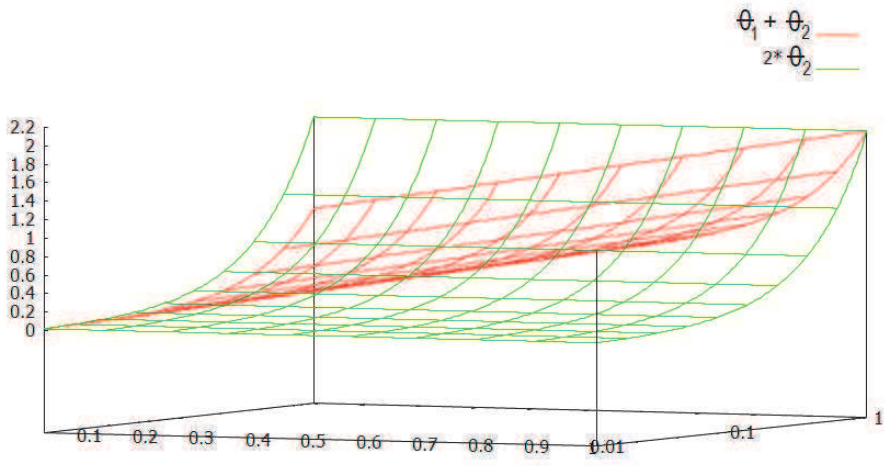


Figure 4.5: The two cost functions $\mathcal{F}(H_c, H'_c)$ and $\mathcal{F}(H_c, H''_c)$

$\mathcal{F}$ is used, the order in which messages are exchanged to obtain localized bMSCs is not significant. Considering that the cost function is influenced only by the number of added messages and added processes, we define $\mathcal{F}(H, \{(I_{M'}, l_{M'})\}_{M \in \mathcal{M}})$ as being the cost of a localization of H that satisfies $I_{M'} = I_{f(M)}$, where $f(M)$ has $l_{M'}$ as leader for every $M \in \mathcal{M}$. The localization problem can be formally defined as follows:

Definition 4.3.1. *Let $H = (I, N, \longrightarrow, \mathcal{M}, n_0)$ be a non-local HMSC, and $\mathcal{F}$ be a cost function. The* localization problem *for $H, \mathcal{F}$ consists in returning solutions $s_1, \ldots, s_k$, where each s_i is of the form $s_i = \{(I_{M'}, l_{M'})\}_{M \in \mathcal{M}}$ such that $\mathcal{F}(H, \{(I_{M'}, l_{M'})\}_{M \in \mathcal{M}})$ is minimal, and where for each $M \in \mathcal{M}$, $I_{M'} \subseteq I$ is a set of instances appearing in $M' = f(M)$ and $l_{M'} \in I_{M'}$ is the leader of M'.*

4.4 Localization as a constraint optimization problem

This section explains how a finite domain constraint optimization model is constructed from a given HMSC, to minimize the cost of the localization.

4.4.1 Constraint solving over finite domains

A *constraint solving problem* is composed of a finite set of *variables* $X_1, \ldots, X_n$, where each variable X_i ranges over a finite *domain*, noted $D(X_i)$. An assignment of a variable is a choice of a value from its domain. A set of *constraints* $C_1, \ldots, C_m$ is defined over the variables and the goal in a constraint solving problem is to find *solutions*, i.e., assignments for all variables, that satisfy all constraints. A constraint solving problem is *satisfiable* if it allows at least one solution. When a *cost function* $\mathcal{F}$ is associated to each assignment, the problem becomes a *constraint optimization problem* (COP) where the goal is to find a solution that optimizes the cost. Such a solution is called an *optimal solution*. Constraint solving frequently uses filtering and propagation. Roughly speaking, the underlying idea is to consider each constraint in isolation, as a filter over the domains. *Filtering* a domain means eliminating inconsistent values w.r.t. a given a constraint. For example, if $D(X) = \{1, 3, 4\}$ and $D(Y) = \{2, 3, 4, 5\}$, the constraint $X > Y$ filters $D(X)$ to $\{3, 4\}$ and $D(Y)$ to $\{2, 3\}$. Once a reduction is performed on the domain of a variable, *constraint propagation* awakes the other constraints that hold on this variable, in order to propagate the reduction.

Constraint propagation is a polynomial process: It takes $O(n * m * d)$ where n is the number of variables, m is the number of constraints and d is the maximum number of possible values in the domains. Constraint propagation and filtering alone do not guarantee satisfiability, and just prune the domains without trying to instantiate variables. For example, considering the constraint system shown above, the constraint $X > Y$ prunes the domains $D(X)$ to $\{3, 4\}$ and $D(Y)$ to $\{2, 3\}$ but $(3, 3)$ is not a solution of the constraint. The constraint system may even be unsatisfiable, while constraint propagation and filtering does not detect it (i.e., they ensure only *partial satisfiability*). Hence, an additional step called *labeling search* is needed to exhibit solutions. Labeling search consists in exploring the search space composed of the domains of uninstantiated variables. Interestingly, a labeling procedure can awake constraint propagation and filtering, allowing an early pruning of the search space. In the previous example, if X is labeled by 3 then the constraint $X > Y$ is awoken and automatically reduce the domain of Y to $\{2\}$. A labeling search procedure is *complete* when the whole search space is explored.

Complete labeling search can eventually determine satisfiability (or unsatisfiability) of a constraint solving problem over finite domains. However, it is an exponential procedure in the worst case. This is not surprising as determining satisfiability of a

constraint problem over finite domains is NP-hard [85].

During labeling search, when a solution s is found, the value $m = \mathcal{F}(s)$ of the cost function can be recorded, and backtracking can then be enforced by adding the constraint $\mathcal{F}(...) < m$ to the set of constraints (or $\mathcal{F}(...) \leq m$ if one wants to explore **all** optimal solutions). If another solution is found, then the cost function $\mathcal{F}$ will necessarily have a cost smaller than m. This procedure, called branch&bound [60], can be controlled by a timeout that interrupts the search when a given time threshold is reached. Of course, the current value of $\mathcal{F}$ in this case may not be a global minimum, but it is already an interesting value for the cost function, something that we call a *quasi-optimum*.

For localization of HMSCs, selecting the local HMSC with the smallest cost is desirable but not always essential. On the other hand, mastering the time spent for localization is essential to scale to real-size problems.

4.4.2 From HMSC to COP

To simplify notations, we will consider that in the HMSCs we consider, all nodes are either initial nodes, end nodes or choice nodes. Any HMSC can be transformed in an equivalent HMSC of this kind.

Variables. Localizing an HMSC H consists in selecting a set of participating instances and a minimal process for each bMSC appearing in H, such that every choice in the HMSC becomes a local choice. As this selection is not unique, we use constraint optimization techniques to provide characteristics of localized HMSCs with minimal cost. We propose to transform any HMSC into a constraint optimization problem, as follows: A couple of variables (X_i, Y_i) is associated to each bMSC $M_i \in \mathcal{M}$, where X_i represents the set of instances chosen for the bMSC $f(M_i)$, and Y_i represents the leader in $f(M_i)$. If I is the set of instances of H, every X_i takes its possible values in 2^I while Y_i takes a value in I.

Constraints.

Our constraint model is composed of domain, equality and inclusion constraints. *Domain constraints*, noted DOM, are used to specify the domains of X_i and Y_i. Obviously, if a bMSC M_i is defined over a set of processes $\mathcal{P}_i$, we have $\mathcal{P}_i \subseteq X_i \subseteq I$. *Equality constraints*, noted EQU, enforce the locality property. For two bMSCs M_i, M_j such that there exists two transitions (n, M_i, n_1) and (n, M_j, n_2) in $\rightarrow$ originating from the

same node n, $f(M_i)$ and $f(M_j)$ must have the same leader, i.e., $Y_i = Y_j$. We write $M_i \otimes M_j$, when such choice between M_i and M_j exists in H. Locality of HMSCs is also enforced by using *inclusion constraints*, noted $INCL$. Let $M_i, M_j \in \mathcal{M}$ be two bMSCs. We write $M_i \triangleright M_j$ when there exists a path $(n, M_i, n')(n', M_j, n'')$, i.e., when M_i is the predecessor of M_j in H. In such case, in any localization of H, the minimal instance of $f(M_j)$, represented by variable Y_j, must appear in the set of instances of $f(M_i)$, represented by variable X_i. In our constraint model, this is expressed by the constraint $Y_j \in X_i$. Similarly, the leader of a bMSC in the localized solution can only be one of its instances, so we have $Y_i \in X_i$ for every $M_i \in \mathcal{M}$.

It is worth noticing that the localization problem is always satisfiable, as there exists at least one trivial solution: Select an instance in I as leader for all bMSCs, then add this instance if needed to every bMSC, and messages from this instance to all other instances. However, this trivial and uninteresting solution is not necessarily minimal w.r.t. the chosen cost function.
We can now prove that our approach is *sound* and *complete* by considering the following definition:

Definition 4.4.1. *Let $H = (I, N, \rightarrow, \mathcal{M}, n_0)$ be an HMSC, the constraint optimization model associated to H is $CP_H = (\mathcal{X}, \mathcal{Y}, \mathcal{C})$ where $\mathcal{X} = \{X_1, \ldots, X_{|\mathcal{M}|}\}$ associates a variable to the set of instances appearing in each bMSC of $f(\mathcal{M})$, $\mathcal{Y} = \{Y_1, \ldots, Y_{|\mathcal{M}|}\}$ associates a variable to the leader selected for each bMSC of $f(\mathcal{M})$, and $\mathcal{C} = DOM \cup EQU \cup INCL$ is a set of constraints defined as follows:*

- $DOM = \bigwedge_{i \in 1 \ldots |\mathcal{M}|} X_i \in 2^I \wedge \mathcal{P}_i \subseteq X_i \wedge Y_i \in I$;
- $EQU = \bigwedge_{M_i, M_j | M_i \otimes M_j} Y_i = Y_j$
- $INCL = \bigwedge_{M_i, M_j | M_i \triangleright M_j} Y_j \in X_i \wedge \bigwedge_{i \in 1 \ldots |\mathcal{M}|} Y_i \in X_i$

Then, solving the localization problem for an HMSC H amounts to find an optimal solution for CP_H, w.r.t. cost function $\mathcal{F}$. We have:

Theorem 4.4.1. *Computing solutions for a localization problem using an optimal solution search for the corresponding COP is a sound and complete algorithm.*

This result is not really surprising, as CP_H represents what is needed for an HMSC to become local. For the sake of completeness, a formal proof of this theorem is available in appendix 7.2.

4.5 Implementation and experimental results

To evaluate the approach proposed in the chapter, we implemented a systematic transformation from HMSC descriptions to COPs and conducted an experimental analysis over a large number of randomly generated HMSCs. This effort is justified by the absence of such important collection of problems or HMSCs that will allow a significant experimental analysis. Our implementation contains three main components G, A and S and is described in Figure 4.6. G is a random HMSC generator, A is an analyzer that transforms a localization problem for a given HMSC into a COP, as described in the previous section. Finally S is a constraint optimization solver: We used the `clpfd` library of SICStus Prolog [21].

The generator G takes an expected number of distinct HMSCs to generate (nbH), a number of bMSCs in each HMSC(nbB), and a number of active processes in each HMSC (nbP) as inputs. As output, it produces an xml file containing nbH randomly generated HMSCs (Check the appendix for further details about the generator).

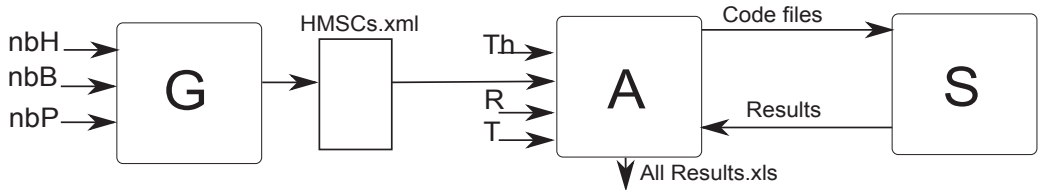


Figure 4.6: The input and outputs of the generator, the analyzer and the solver.

The analyzer A takes Th, the set of parameters θ_1 and θ_2 of the messages and processes counting cost function $\mathcal{F}$ (defined in section 4.3), a sequence of time-out values T, the generated HMSCs (the xml file $HMSCs.XML$) as inputs, and a set of heuristics R such that a heuristic is a rule or a strategy that we use to make a good decision. It provides a shortcut to solving difficult problems (A simple example of heuristic is deciding to use a toothpaste A rather than toothpaste B only because A is more expensive).

The generator's output xml file will be the first entry of the analyzer as presented in figure 4.6. The aim of this analyzer is to help us to decide for a simple and efficient way to resolve our localization problem, and to evaluate the influence of some parameters on the runtime execution. To solve the problems, and so to localise the generated HMSCs, the analyzer transforms those generated partial HMSCs into COPs. To solve

the COPs, the analyzer uses a well known solver named *SICStus Prolog*. Prolog is a well-established declarative and high-level programming language, besides it is a simple but powerful constraint logic programming language. Prolog have many developement environment and one of them which is the *SICStus Prolog* developement system [21]. *SICStus Prolog* is used in a wide range of domains (Speech applications, Telecom, Biotech,Logistics, Data Mining , ...) and by many important customers(the NASA Intelligent Systems Division, Ericsson AB, Pyrosequencing AB, RedPrairie, ...) and it have proven its efficiency. The analyzer generates, for each CSP, a prolog file that defines the problem (variables, domains and constraints) with a constraint model implemented in $clpfd$(Constraint Logic Programming over Finite Domains), a library of the SICStus Prolog environment. The $clpfd$ library is an important reason for us to choose Prolog, since it allows us to benefit from a very efficient implementation of constraint propagation [21]. The constraint propagation consists in explicitly forbidding values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise. The idea behind this technique is to reduce the search tree so that the backtrack search commits into less inconsistent instantiations. The clpfd library contains also the branch and bound algorithm that we are planning to use.

As presented in figure 4.6, the analyzer takes a second entry R that represents the set of heuristics that we might want to apply to solve the COPs. To run a COP with a given heuristic the analyzer first edits this heuristic in the prolog file (named "Code files" in Figure 4.6) of the COP then it calls the solver to compile and run the prolog code. When the execution ended the results (the solution, the time of execution, ...) are edited in a text file (named "Results" in Figure 4.6). The analyzer reads these results and save them to be collected with the other results that it might get later. All the results are then collected in the file "All Results.xls" presented in Figure 4.6. In the experiments, we considered several labeling heuristics to choose the variable and the value to enumerate first, e.g., *leftmost*, *first-fail*, *ffc*, *step* or *bisect*. *Leftmost* is a variable-choice heuristic that selects the first unassigned variable from a statically ordered list. *First-fail* is a dynamic variable-choice heuristic that selects first the variable with the current smallest domain. *Ffc* is an extension of *first-fail* that uses the number of constraints on a given variable as a tie-break when two variables have the same domain size. *Step* is choice-value heuristic that consists in traversing incrementally the variation domain of the current variable. Finally, *bisect* implements *domain splitting* which consists in dividing a domain into two subdomains and propagating the subdomain information. For example, if x takes a value in an interval $[a, b]$, then

bisect will propagate first $x \in [a, \frac{a+b}{2}]$, and then $x \in [\frac{a+b}{2}, b]$ upon backtracking. For each generated HMSC H and each heuristic $h_i \in R$, the analyzer creates a prolog file that contains the corresponding COP and the heuristics to apply during search. For efficiency reasons, a special attention has been paid to the encoding of variation domains and constraints. Subset domains were encoded using a binary representation and sets inclusion using efficient div/mod operations.

Another entry of the analyzer, the sequence T represents the various instants at which the optimization process must temporarily stop, and return the current best value found for the cost function. These values are quasi-optima, representing approximations of the global optimum. The combination between heuristics and time-out values is useful to compare different labeling strategies. Finally, the analyzer A collects all the results returned by the solver with the time needed to provide a solution, and stores them for a systematic comparison.

The first step of the experiment consisted in a systematic evaluation of the performance of several heuristics to guide the solver. During this step, we considered several heuristics and time-outs. We do not report here all the results, but show only the results for one representative model of 7 bMSCs with 7 instances (We chose this model randomly between many others just to show how the results may be affected by the heuristics). Figure 4.7 shows the time-aware minimization of the cost value with 12 different heuristics and time-outs between 1s and 14s for a chosen localization problem. Heuristics descriptions use the following syntax: *[b | u] / [left | ff | ffc] / [bisect | step] / [XYC | YXC | CXY | CYX]*, where b and u stand resp. for *bounded costs* and *unbounded costs*. The heuristic *bounded cost* evaluates a lower bound *Lowb* on the cost of a solution that can be reached from a given state. We can compute the lower bound *Lowb* on the cost of a solution as we know that for each bMSC M_i we need minimum $k_i - 1$ messages to establish the localization, where k_i is the number of minimal processes in M_i, then *Lowb* is computed as follows:

$$Lowb = \sum_{i \in |M|} (k_i - 1)$$

Left, *ff* and *ffc* stand for a variable-choice heuristic, *bisect* and *step* stand for value-choice heuristic, and *XYC*, etc. stand for the static order in which variables are fed to the solver. Bold values indicate *proved global minima*, non-bold values indicate *quasi-optima*, – indicates absence of result in the given time contract.

	heuristics \ runtime	1 s	2 s	3 s	4 s	5 s	6 s	7 s	12 s	13 s	14 s
0	u/left/step/XYC	23	23	22	22	22	22	19	19	19	19
1	u /left/step/YXC	19	**19**								
2	b/left/step/XYC	19	**19**								
3	b/left/step/YXC	**19**									
4	b/left/step/CYX	-	-	-	-	-	-	-	-	21	**19**
5	b/ff/step/XYC	22	19	19	19	19	**19**				
6	b/ff/step/YXC	30	19	19	19	19	**19**				
7	b/ff/step/CYX	30	19	19	19	19	**19**				
8	b/ffc/step/CYX	27	22	19	19	19	19	**19**			
9	b/left/bisect/XYC	**19**									
10	b/left/bisect/YXC	**19**									
11	b/left/bisect/CYX	-	-	-	-	21	**19**				

Figure 4.7: Comparing heuristics with one representative example.

In Figure 4.7, heuristics 3,9,10 give the best results as they find the optimal solution in less than one second. The series of experiment that we run on many test examples shows that heuristics with an estimation of cost, and a static ordering of variable evaluations have the best performance. Overall, the heuristics number 10 combining *bisect* (domain splitting), *left* (static variable ordering), and a cost evaluation exhibited the best results based on the statistics of many test examples we have run. We selected this heuristic for the next steps of the experiment.

As next steps, we generated 11 groups of 100 random HMSCs each. Each generated HMSCs contains 10 bMSCs, which is a reasonably large number, according to the existing literature on HMSCs. We then let the number of processes grow from 4 to 14. We also generated 12 groups of 100 HMSCs, containing exactly 8 processes, and let the number of bMSCs grow from 4 to 15. The goal of these series of experiments was to evaluate the influence of both the number of processes and the number of bMSCs on the runtime of our localization approach. We expected these parameters to influence the performance of localization, as increasing the number of bMSCs increases the number of variables, and increasing the number of processes increases the size of variables' domains. We have obtained solutions for all HMSCs, which allowed us to evaluate the impact of both parameters. The evaluation was performed on a machine equipped with INTEL $P9600$ core2 Duo at $2,53$ Ghz, with $4Go$ of RAM. Results of both experiences are given in Figure 4.9 and Figure 4.10, using box-and-whiskers plots to show the statistical distribution of datasets.

The Box-and-Whiskers is an exploratory graphic used to show the distribution of a dataset. The Figure 4.8 shows how to read it.

- The *outliers* presents problems where data where more(/less) than 1.5 times of upper(/lower) quartile.

- The *Maximum* is the greatest value, excluding outliers.
- The *Upper Quartile*: 25 percent of data are greater than this value.
- The *Median*:50 percent of data are less than this value.
- The *Lower Quartile*: 25 percent of data are less than this value.
- The *Minimum* is the least value excluding outliers

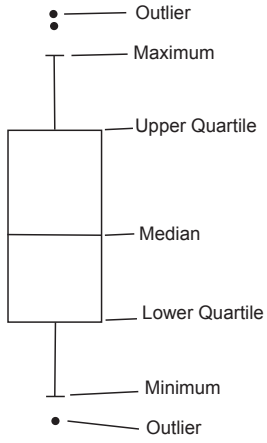


Figure 4.8: A Box and Whisker Plot

Both plots in Figure 4.9 and Figure 4.10, use logarithmic scales to tackle the big variance between runtime measurements. The horizontal axis represents Box-and-Whiskers plot of each group of problems. As expected, the plots show exponential curves but the runtime for each group remains quite low. For randomly generated HMSCs of reasonable size (such as the ones found in the literature), our experimental results show that localization using constraint optimization takes a few minutes in the worst cases, and an average duration of a few seconds. Actually, even for the largest cases (15 bMSCs with 14 processes), the runtime of our localization approach did not exceed 40 minutes. Although solving COPs over finite domains is NP-hard [85], as examples of existing HMSCs usually contain less than 15 bMSCs, our localization process appears to be of practical interest. Our results are encouraging, and show that the approach is fast enough to be used in practice. However, experimental results have been obtained on random instances only, and thus further experiments on non-random instances are necessary to confirm this judgment.

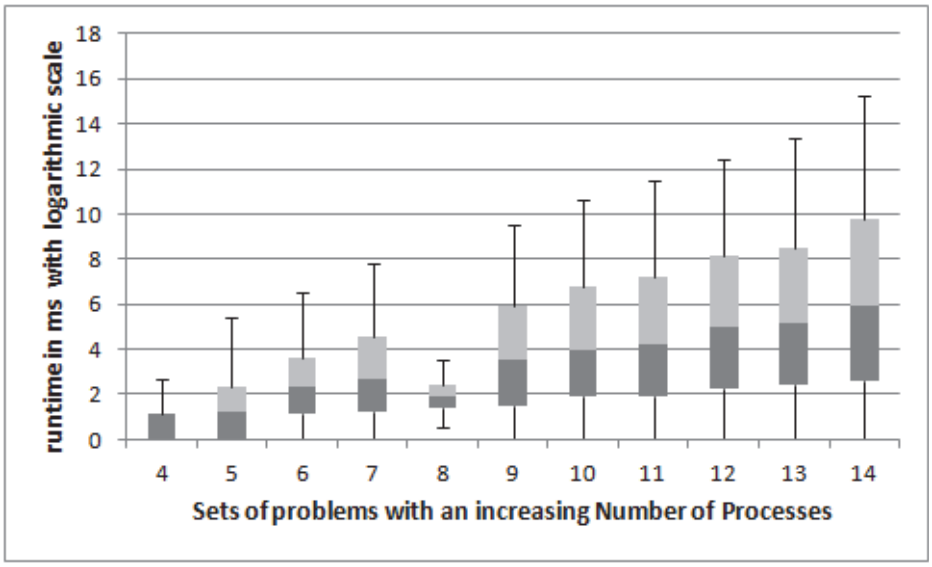


Figure 4.9: Influence of the number of processes on the runtime execution

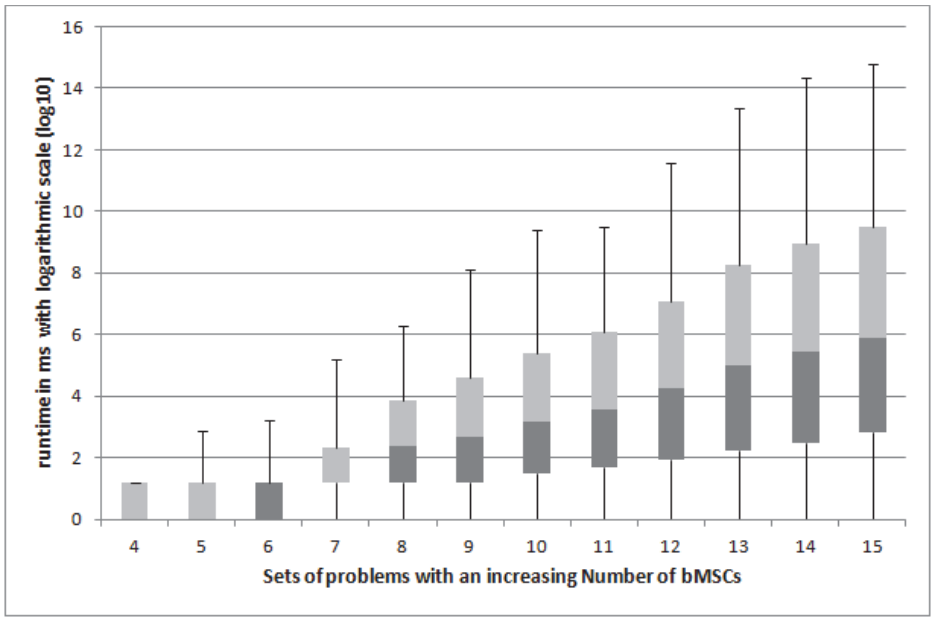


Figure 4.10: Influence of the number of bMSCs on the runtime execution

4.6 Conclusion and future work

In this chapter, we have proposed a sound and complete method to transform arbitrary HMSCs into implementable ones. We first generate a constraint optimization problem from the non-local HMSC.

The solution returned by a solver can be used to build an optimal localized version of the original specification, without changing the overall architecture of the HMSC. Once an HMSC is localized by addition of messages and processes in bMSCs, automatic implementation techniques can generate code for communicating processes.

This approach has been implemented and tested on a benchmark of 2300 randomly generated HMSCs. The experimental results show that our approach is of practical interest: It usually takes less than a few minutes to localize an HMSC.
There are four foreseen extensions of this work. First, other cost functions can be considered as our approach does not depend on the choice of a particular cost function (For instance, localization with cost functions that accounts for the cost of communications between instances). Second, we plan to allow modifications of the HMSC, in addition to those brought to the bMSCs of the specification. Considering architectural constraints that disallow communications between some processes is another challenging issue, as in this case existence of a solution is not guaranteed. Finally, noticing that localization is a rather syntactic procedure, the question of designating a process as a leader or adding messages should also be addressed in more semantics terms. Further work also includes the experimentation of our approach on industrial case studies, to evaluate its performance on non-random HMSCs. We have started a collaboration with a company that develops communicating systems and wants to generate test cases based on requirements design. Our approach will be useful to derive automatically test cases from HMSCs that were designed without any requirement on implementability.

Chapter 5

SOFAT tool

In the previous chapters, we have presented two approaches: first the correct synthesis of any local HMSC, and then the localization of a non-local HMSC. With these two algorithms, any HMSC is either implementable, or can be transformed in a correctly implementable HMSC with minimal changes to the original specification. In this chapter, we present the implementation work related to these approaches.

We have implemented these approaches to allow the use of the algorithms in practice, and all these functionalities are added into an existing tool called SOFAT presented in section 5.1. Then, in section 5.2, we give a use case example presenting a simple transmission protocol in a distributed system based on the Morse Code. Then, in sections 5.3, 5.4, and 5.5, we present respectively the implementation of the synthesis algorithm that allows the projection of an HMSC into a CFSM with controllers and tagged messages, the code generation of the corresponding Promela code, and the code generation of a JAVA code for a REST platform. In section 5.6, we present the implementation of the localization procedure.

5.1 Description of SOFAT

SOFAT [36] is the acronym for Scenario Oracle and Formal Analysis Toolbox. SOFAT allows the edition and analysis of distributed systems specifications described using Message Sequence Charts. It is a formal toolbox for manipulation of scenarios.

The main functionalities proposed by SOFAT are the textual edition of Message Se-

quence Charts, their graphical visualization, the analysis of their formal properties, and their simulation. The analysis of the formal properties of a Message Sequence Chart specification determines if a description is regular, local, or globally cooperative. Satisfaction of these properties allow respectively for model-checking of logical formulae in temporal logic, implementation, or comparison of specifications. Classifying an HMSC according to its properties is important, as a choosen application might be an undecidable problem in general, but become tractable for some sub-classes of HMSCs. The SOFAT toolbox implements most of the theoretical results obtained on Message Sequence Charts this last decade. The purpose of this software is twofold:

- Provide a scenario based specification tool for developers of distributed applications
- Serve as a platform for theoretical results on scenarios and partial orders

SOFAT provides several functionalities, that are: syntactical analysis of scenario descriptions, formal analysis of scenario properties, interactive simulation of scenarios (when possible) [59], and diagnosis from scenario models (starting from an HMSC and an observation collected by partially monitoring a system, find all runs of the HMSC that can explain the observation).

In this thesis, we have extended SOFAT with code synthesis functionalities, allowing to generate communicating automata, promela code, rest based web services from HMSCs, and the localization procedure. Next we will detail each of these new functionalities.

5.2 Use Case

In this section, we design a case study, namely a simple transmission protocol in a distributed system based on the Morse Code.

Morse code was invented by Samuel F. B. Morse and his assistant Alfred Vail in 1832. It allows to transmit information by using a sequence of dots and dashes that represent letters. To achieve a high efficiency of information transmission, the inventors of the Morse code assigned shorter sequences of dots and dashes to the more frequently used characters in English and longer sequences to the less common English characters. Figure 5.1 shows the graphical description of the Morse code example HMSC $H1$, and the textual description is given in Figure 5.2. This HMSC is a local HMSC

and we can test its locality.

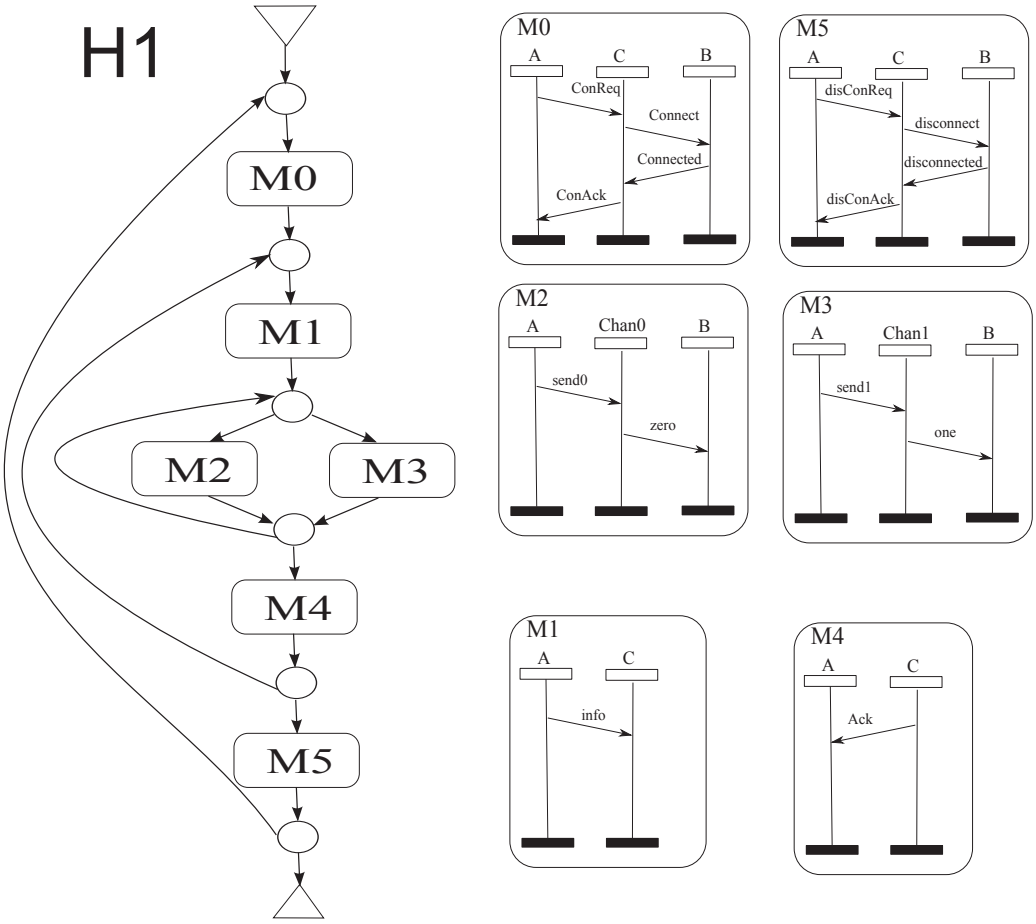


Figure 5.1: The Morse code example HMSC and its bMSCs

In $H1$, the process A wants to transmit Morse coded information to the process B. A proceeds by requiring a connection to B via the Morse Coder C (bMSC $M0$). Once the connection is established A sends the information to the Morse Coder C (bMSC $M1$). C translates the information received from A into a sequence of binary digits 0s and 1s (bMSC $M2$) that we will use to 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. We will consider that all the 0s are sent via the channel $chan0$ (bMSC $M3$) and all the 1s are sent via the channel $chan1$ (bMSC $M4$). 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 information or to close the

```
mscdocument MorseCodeExample;

    msc MorseCode ;
          expr L1;
          L1:M0 seq(L2);
          L2: connect seq  (L3);
          L3: M1 seq (L4) ;
          L4: connect seq  (L5 alt L6);
          L5:M2 seq (L7);
          L6:M3 seq (L7);
          L7:connect seq(L8 alt L4);
          L8:M4 seq(L9);
          L9:connect seq(L2 alt L10);
          L10:M5 seq (L11);
          L11:connect seq(L12 alt L1);
          L12:end;
    endmsc;

    msc M0;
        inst A,C,B ;
        instance A;
          out ConReq to C;
          in ConAck from C;
        endinstance;
        instance C;
          in ConReq from A;
          out Connect to B;
          in Connected from B;
          out ConAck to A;
        endinstance;
        instance B;
          in Connect from C;
          out Connected to C;
        endinstance;
    endmsc;

    msc M1;
        inst A,C ;
        instance A;
          out info to C;
        endinstance;
        instance C;
          in info from A;
        endinstance;
    endmsc;

    msc M2;
        inst C,Chan0,B ;
        instance C;
          out send0 to Chan0;
        endinstance;
        instance Chan0;
          in send0 from C;
          out zero to B;
        endinstance;
        instance B;
          in zero from Chan0;
        endinstance;
    endmsc;

    msc M3;
          inst C,Chan1,B ;
          instance C;
            out send1 to Chan1;
          endinstance;
          instance Chan1;
            in send1 from C;
            out one to B;
          endinstance;
          instance B;
            in one from Chan1;
          endinstance;
    endmsc;

    msc M4;
          inst A,C ;
          instance A;
            in Ack from C;
          endinstance;
          instance C;
            out Ack to A;
          endinstance;
    endmsc;

    msc M5;
          inst A,C,B ;
          instance A;
            out disConReq to C;
            in disConAck from C;
          endinstance;
          instance C;
            in disConReq from A;
            out disconnect to B;
            in disconnected from B;
            out disConAck to A;
          endinstance;
          instance B;
            in disconnect from C;
            out disconnected to C;
          endinstance;
    endmsc;


endmscdocument;
```

Figure 5.2: The textual description of the Morse code example of Figure 5.1

connection. Figure 5.5 presents the projection of the HMSC $H1$ on all the instances of the system. The method used in the projection is the one described in chapter 3. Each process interacts with its controller and controllers interact as presented in Figure 5.3. The Figure 5.4 presents an example of a message sending of a message $m1$ from an automaton A into the automaton B via C_A and C_B the respective controllers. Note that these automata can be generated automatically by SOFAT as well. We will give a number to each process going from 0 to 4 associated respectively to the processes A, B, C, Chan0 and Chan1.

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 $ConReq$, then A sends to its controller the message $\{2, ConReq, M1\}$ that corresponds to the $\{i, m, j\}$ structure described in the algorithm. The 2 in this message corresponds to the destination process, so when the controller receives the messages it knows to which controller it should be sent (the process number 2 is C). $ConReq$ represents simply the data part of the message that should be sent by the controller, $M1$ means that the process A wants to execute a new branch that begins with bMSC $M1$, so the controller have to update its local tag. The new tag is concatenated to the message that will be sent.

In the example of execution presented in Figure 5.6, we can see the execution of the sequence : $M0$, $M1$, $M2$ and finally $M3$. In $M0$ the connection is established between the processes A and B via C, then process A sends to C the information to be coded and sent to process B. C codes the information and in the example the information is coded as a sequence of two bits: 0 1. C runs the bMSC $M2$ then the bMSC $M3$. A difference in the performance between $chan0$ and $chan1$ causes a delay and a message overaking in the diagram: the message *one* arrives at the controller of B $Cont_B$ before the message *zero*. When the message *one* tagged with [111000] arrives at the controller $Cont_B$, $Cont_B$ compares this tag with the local tag that is equal to [100000] after the execution of the bMSC $M1$ where B was concerned. The tag [111000] is not the direct successor of the local tag by projection on the components concerning B, so the controller delays the delivery of the message *one* to the automaton B. When the message *zero* arrives at $Cont_B$, the controller of B, $Cont_B$ finds that the tag of this message ([110000]) is the direct successor of the local tag ([100000]) so $Cont_B$ sends the message to B and updates its local tag to [110000]. $Cont_B$ compares again the tag of the delayed message *one* ([111000])with its local tag, now this tag is the direct successor of the local tag so $Cont_B$ sends the message to B and updates its local tag. On this example, one can easily see that the tags allowed for the messages zero, one,

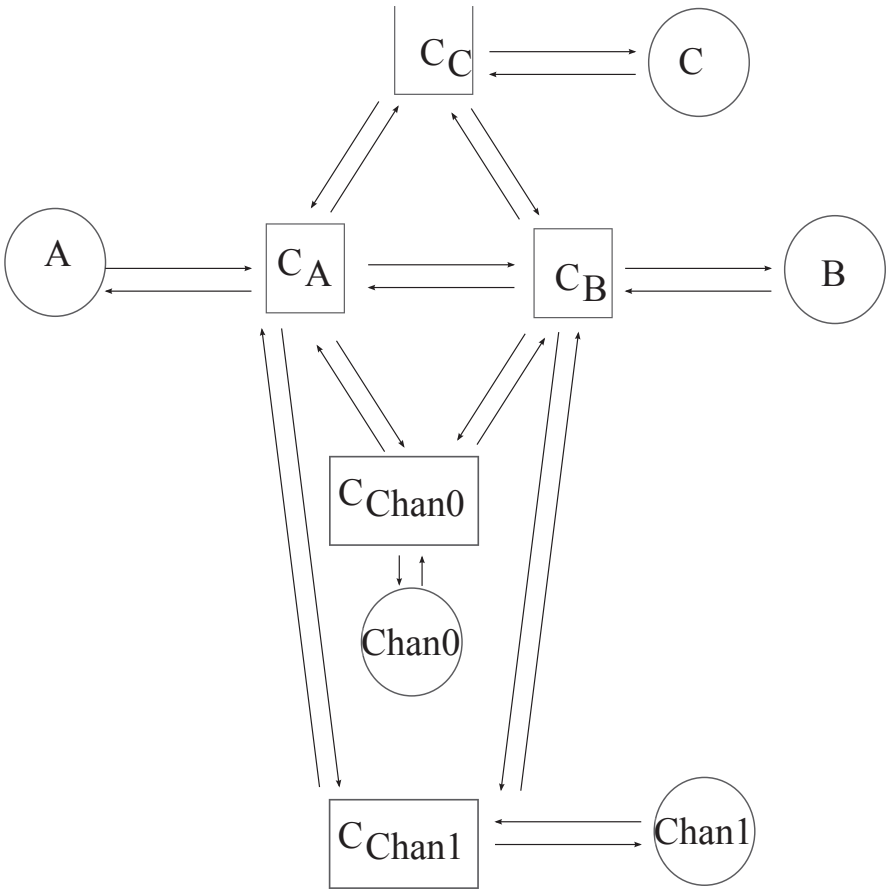


Figure 5.3: The architecture of the Morse code example of Figure 5.1

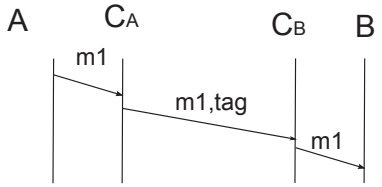


Figure 5.4: Automaton A sends a message m1 to the automaton B via the controllers

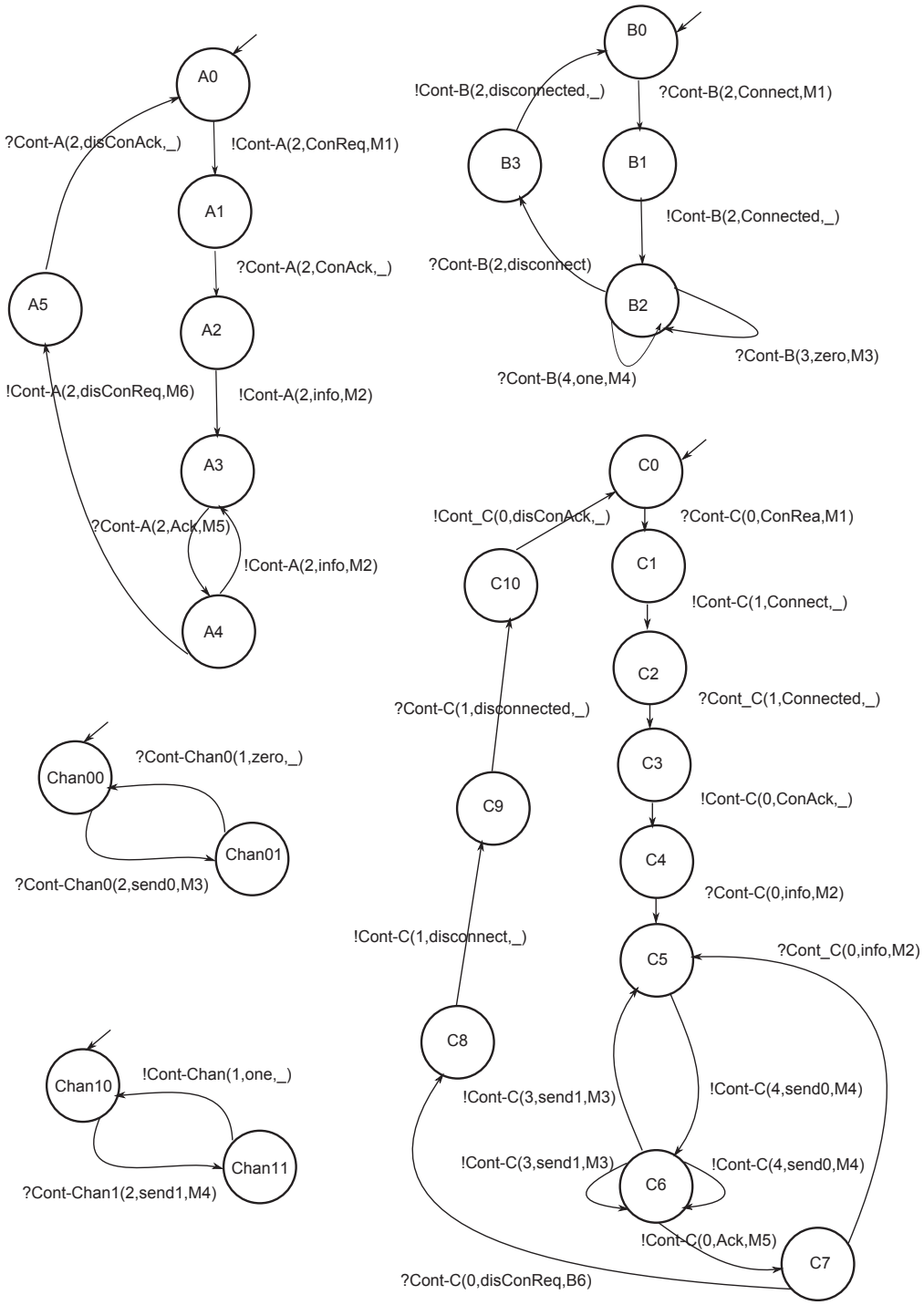


Figure 5.5: The CFSM obtained by projection of the HMSC Figure 5.1

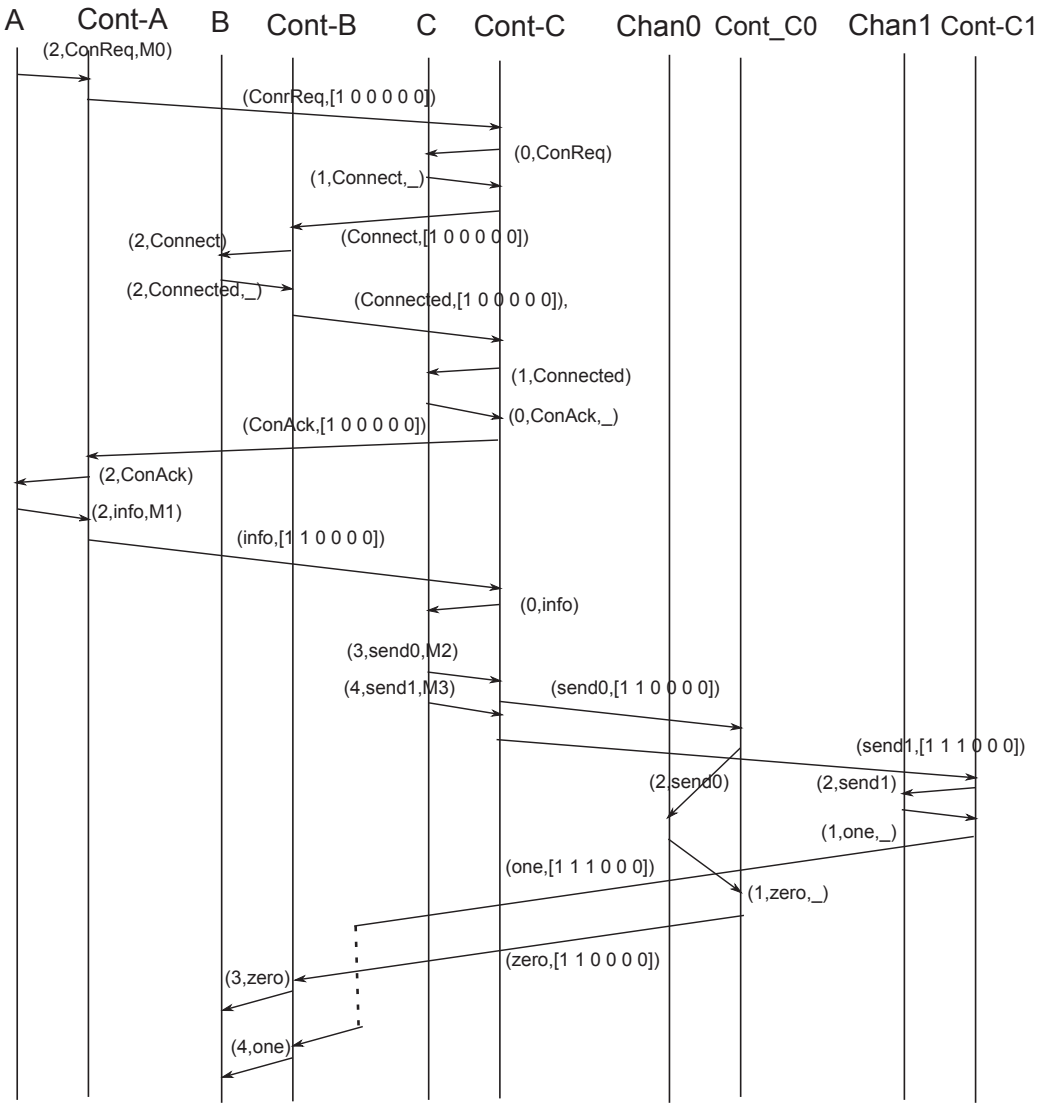


Figure 5.6: An execution of the CFSM of Figure 5.5 with their controllers

to be received in the order choosen by process A.

5.3 CFSM generation

In SOFAT, we have also implemented both projections of an HMSC into a CFSM: the direct projection, and the projection with tags and controllers of the synthesis algorithm introduced in chapter 3. Note that the CFSM is only an abstract model. Once we have the equivalent CFSM, the actual generation of code for simulation tools like XSPIN or for a particular target platform like REST for example is another task. The code generation of Promela and JAVA code will be respectively presented in sections 5.4 and 5.5.

5.4 Promela code generation

In addition to automata synthesis, SOFAT can output the equivalent Promela code. Promela is the input language of the model-checker SPIN [39]. It provides a flexible and abstract view of a distributed system, that we call our "high-level" implementation. In Promela, we can easily describe the distributed architecture and the behavior of each component using the notion of Promela process and guarded commands. Using SPIN, it is then possible to simulate the target code (to simulate an execution of the specification, presented in SPIN by basic MSCs). and also to model-check some properties. The Promela code generated by SOFAT for the Mose Code example of Figure 5.1 is given in the appendix 7.3.1.
Figure 5.7, shows an execution of our Promela code produced by SOFAT and run by XSPIN. This simulation shows clearly the exchanged messages, the different tags stamping the messages and their evolution with respect to the executed choices. It also shows that the algorithm preserves a correct ordering of message receptions.

5.5 Java code generation for Rest platforms

Web services, and more in general service-oriented architectures (SOAs), are emerging among the technologies and architectures of choice for implementing distributed systems for which they provide many fundamental features and benefits.

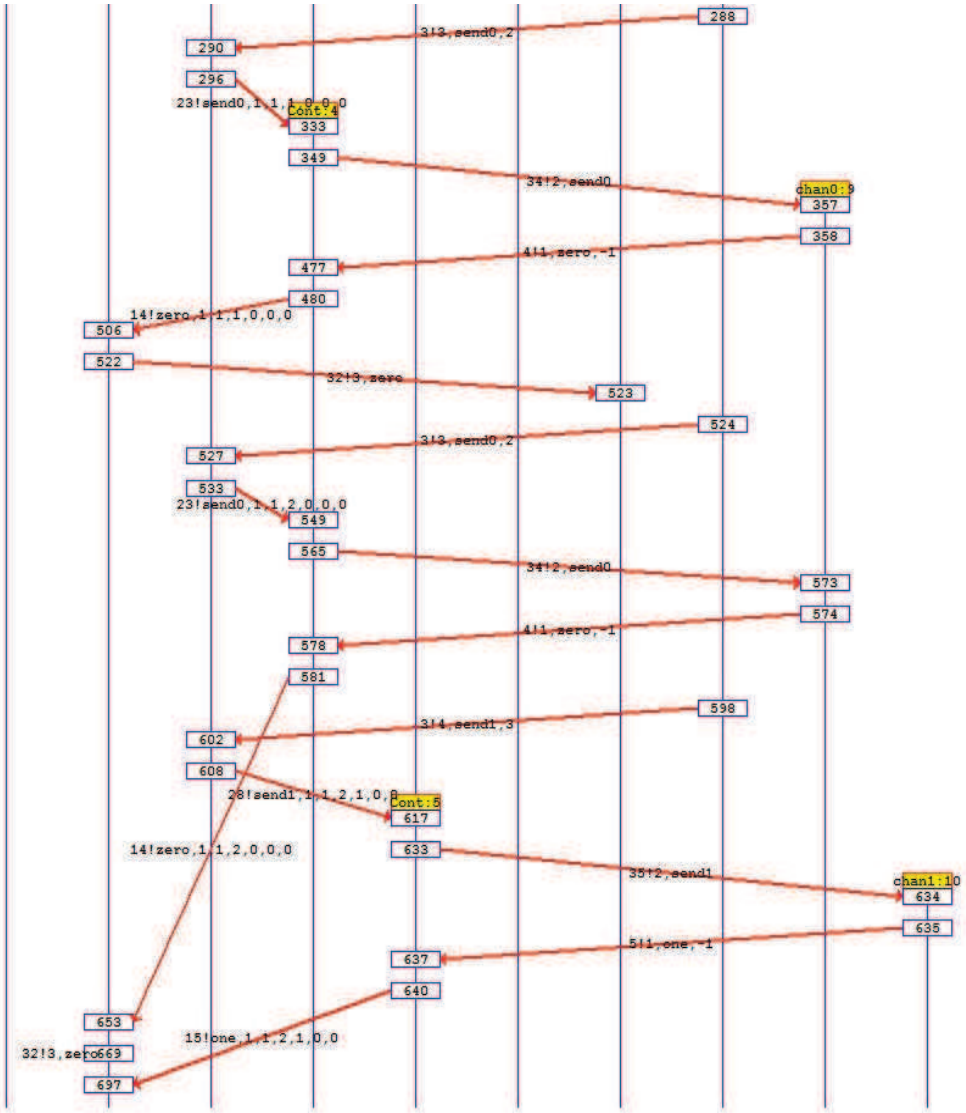


Figure 5.7: An example simulation for the promela code

Web services are software components that communicate using pervasive, standards-based Web technologies including HTTP and XML-based messaging. Their main goal is to enhance the interoperability of distributed systems over networks, especially internet by allowing applications written in different programming languages and running on different platforms to seamlessly exchange data. For further information we refer interested readers to [5] that present a clear state of the art about Web Services.

We choose to generate web services code to be able to run and test our synthesis algorithm in a real application. Besides web services allows us to use communicating machines over the internet instead of creating a specific network, and the generated code is not complicated. We have decided to generate a java code for a REST platforms from HMSCs. REST (Representational State Transfer) designates an architecture style used to create networked applications over the web. The terms "representational state transfer" and "REST" were introduced in 2000 in the doctoral dissertation of Roy Fielding, [28] one of the principal authors of the Hypertext Transfer Protocol (HTTP) [27] specification. REST uses a stateless, client-server, cacheable communications protocol which is almost always the HTTP protocol. Its original feature is to work by using mere HTTP to make calls between machines instead of choosing complex mechanisms such as CORBA, RPC [64] or SOAP [76]. REST presents several advantages we can list:

- REST applies many existing well-known standards (HTTP, XML, URI, and MIME) and need only infrastructure that has already become pervasive.
- HTTP clients and servers are compatible with all programming languages and operating system/hardware platforms, and the default HTTP port 80 is usually left open by default in most firewall configurations. Such lightweight infrastructure, where services can be built with minimal tooling, is inexpensive to acquire.
- REST allows discovering Web resources without any discovery or registry repository.

Note that, we have chosen REST for the advantages that it presents [5] but we could have used SOAP as well to implement our controlled processes.

In the rest of the section, we present the synthesis of REST based services to implement local HMSCs. This synthesis uses as intermediate step the CFSM models synthesized in chapter 3.

5.5.1 Implemented model

Typically, a RESTful Web service define the following aspects: The base/root URI for the Web service (such as http://host/appcontext/resources), the MIME type of the response data supported (which are JSON/XML/ATOM and so on), and the set of operations supported by the service (for example, POST, GET, PUT or DELETE). JAX-RS provides a standardized API for building RESTful web services in Java. The API basically provides a set of annotations and associated classes and interfaces. Jersey, that we used in our implementation, is a reference implementation of JAX-RS [76, 5]. REST uses the HTTP protocol which is synchronous, and so the communications between the different automata and their controllers and between the controllers become synchronous communications. For instance let us consider two machines P

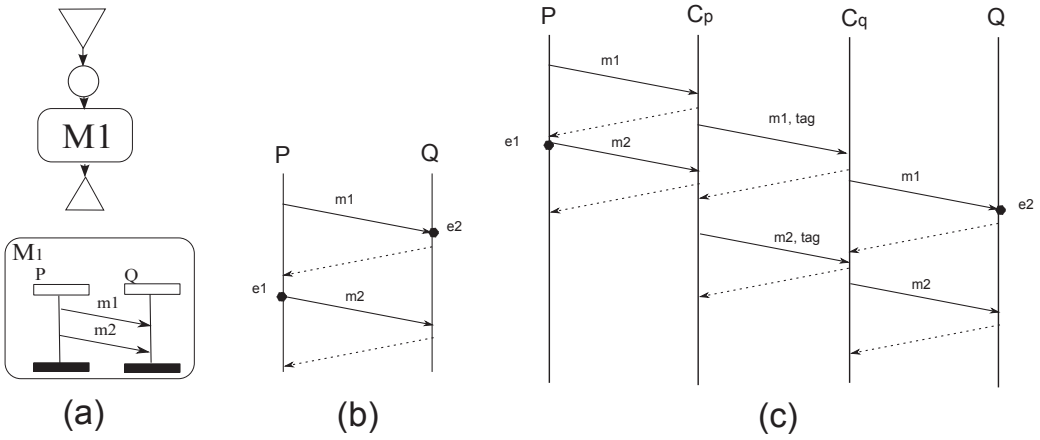


Figure 5.8: Example of run over a Rest platform with two different architectures.

and Q communicating over a Rest platform (using the HTTP protocol), and where the machine P wants to send two requests $m1$ then $m2$ to Q, as defined in the HMSC F of Figure 5.8-a). If we consider the architecture of the Figure 5.8-b) the event $e1$ (which is sending the request $m2$ from P to Q) can not occur before the event $e2$ (which is receiving $m1$) and this is because P should wait for the Acknowledgement from Q that $m1$ has been received before it continues. This is not the case in the architecture with controllers of Figure 5.8-c) where the two events $e1$ and $e2$ are independent. The controllers can be considered as "intelligent buffers", they delay the sending of requests. The controllers allow to preserve the behaviors defined in the original specification (modulo the acknowledgements) without adding new ordering between events. In the architecture of the Figure 5.8-b), which tries to match http invocation and MSCs messages, the sequence of events $e1, e2$ is not allowed. However

it is allowed in the semantics of the HMSC H of Figure 5.8-a)) The acknowledgements will add a small delay to the initial behaviors. In practice, these additional acknowledge messages are transparent for our automata as they occur at the Transport layer (not at the application layer).

Figure 5.9 shows the model that we chose as the implementation model for the code generation. The Automaton is considered as Server with a JFrame client as a graphical interface. The Controller is also a Server. The server of the Automaton is also a client for the Server of the controller. And the sever of the controller is also a client of the server of the other controllers and of the automaton.

Figure 5.10 shows a global view on the architecture and the exchanges of messages and data. One can immediately notice that controllers need not to be placed close to the machines they control in the network to play their role. Automata and controllers are generic programs, that are initialized by reading a description of the automaton they implement, and of the architecture. Automata and controllers are at the same time REST clients and servers.

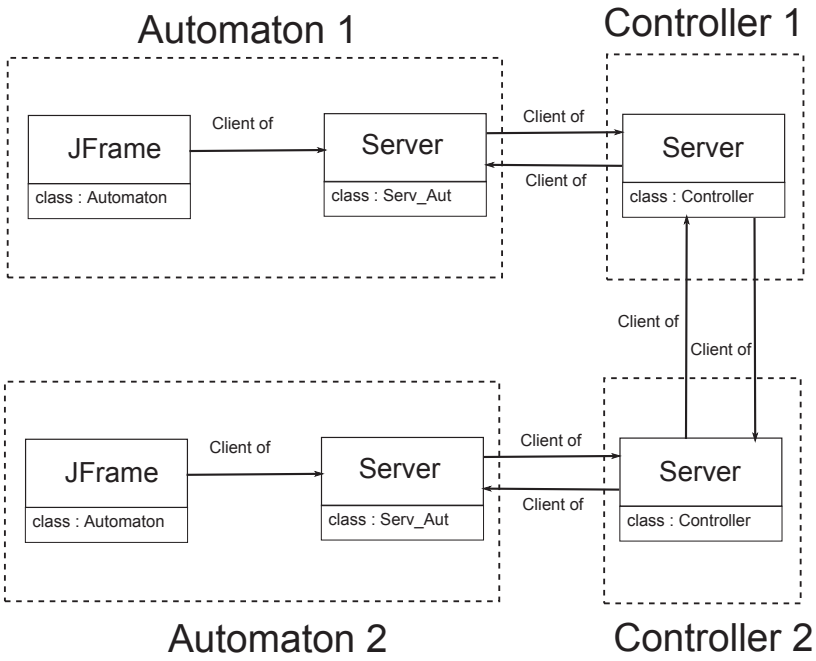


Figure 5.9: The implemented model

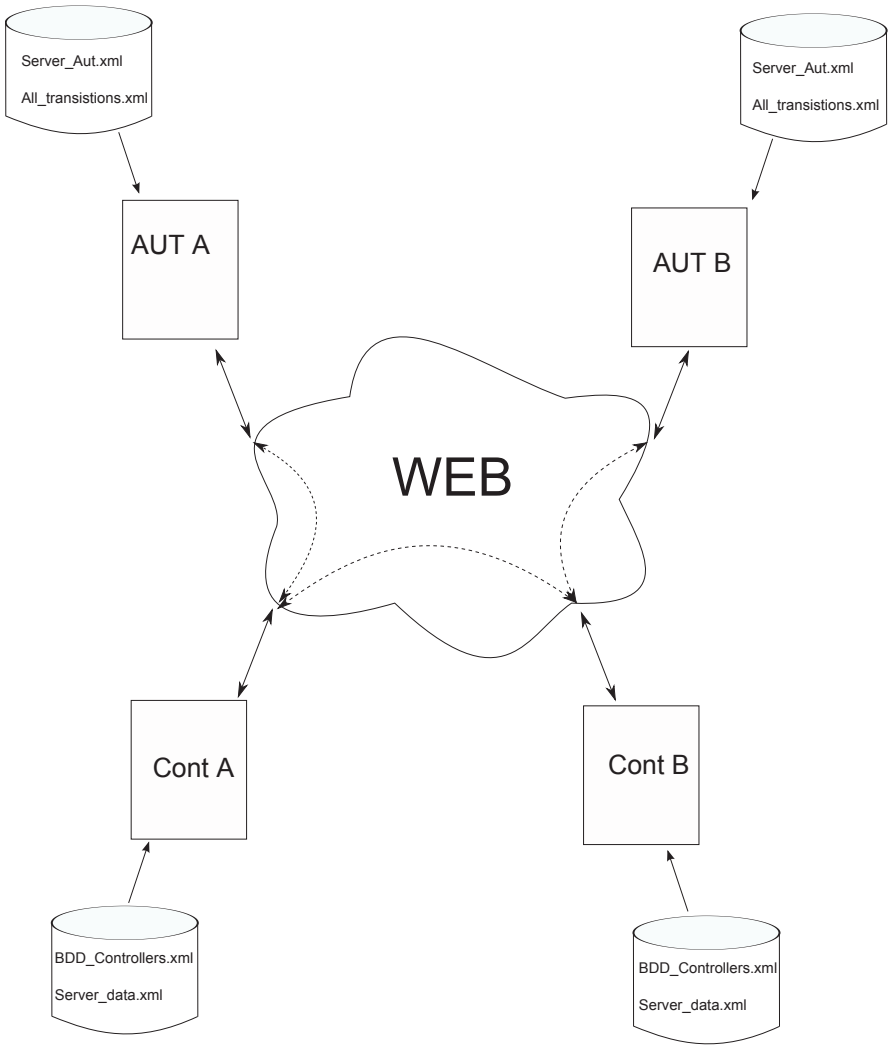


Figure 5.10: Architecture of automata and controllers with REST platform

5.5.1.1 Automaton's generated code

For each automaton we generate an xml file named "all_transitions.xml" that contains the details about this the automaton. It contains a set transitions descriptions that gives the type of the transition (!,?, or local), the origin and the destination of the message, the bMSC or choice that is concerned. An example of a transition is described as follows:

```
<transition>
<number>0</number>
<from>0</from>
<to>1</to>
<type>!</type>
<message>m1</message>
<destination>2</destination>
<choice>0</choice>
</transition>
```

In this example, we have one transition from the state number 0 to the state number 1. This transition consists in sending one a message "m1" to the automaton number 2. This transition is the first action to perform in the branch number 0 of the HMSC from which the automaton was projected.

Another xml file, "Serv_Aut.xml" contains the address of the Server of the automaton itself and that of the Controller.
Figure 5.11 shows the class diagram of the JAVA code generated for each autmaton.

The description of the generated classes for an automaton is as follows:

- "Message" class presents the messages that will be exchanged between the Automaton and its Controller.
- "buffer_Aut" class is the FIFO input buffer of the messages coming from the Controller.
- Both "Transition" and "TransitionHandler" classes are used to read the xml file of transitions of the automaton .
- "AutomatonData" class is used to read the "Serv_Aut.xml" and to get the data about the server.

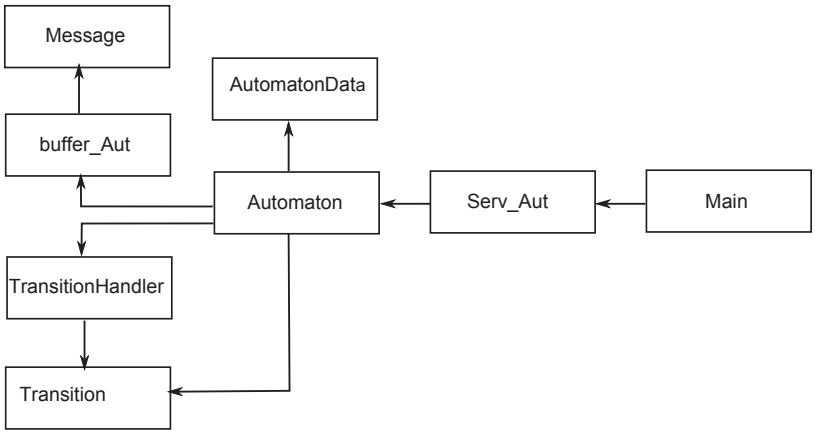


Figure 5.11: Class diagram of the generated code for each Automaton

- "Automaton" class is used to build the graphical interface of the automaton.
- "Serv_Aut" class is the REST Server of the automaton.
- "Main" is the main class.

5.5.1.2 Controller's generated code

As for of the automaton, we generate an xml file for the controller, "Server_data.xml", that contains mainly the address of the Server of the automaton. Another xml file "BDD_Controllers.xml" contains the addresses of all controllers to allow a controller to communicate with its peers.

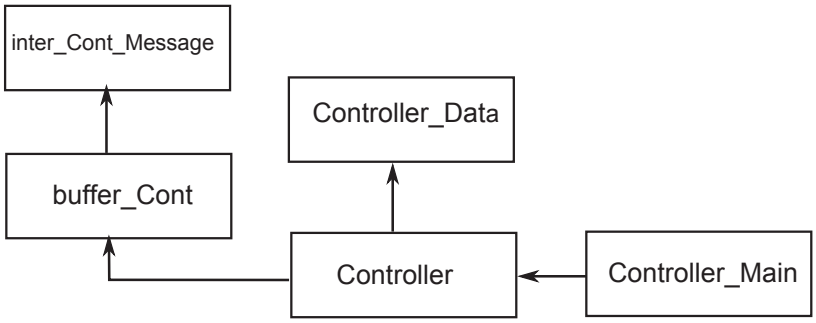


Figure 5.12: Class diagram of the generated code for a Controller

Figure 7.13 shows the class diagram of the JAVA code generated for each controller. The description of the generated classes for a controller is as follows:

- “buffer_Cont” is the buffer of the controller
- “inter_Cont_Message” class is the messages’type that will be exchanged between Controllers
- “Controller_Data” class is used to read the “Server_data.xml” and to get the data about the server
- “Controller” class is the REST Server of the automaton
- “Controlle_Main” is the main class.

The generated code is the same for all the automata and it is the same for the controllers. The major changes are the global architecture of the synthesized REST system, which is driven by the original specification, and then the xml files that are generated by SOFAT. A standard synthesis procedure is as follows:

- SOFAT first read the HMSC textual description entered by the user,
- When the user choose the synthesis of JAVA code for REST Platform:
  - the abstract model is created
  - the addresses of the servers are entered via a popup window before generating the code for the Rest platform. Figure 5.13 shows this graphical interface for the Morse code example.
  - once the addresses are validated, the xml files and the code are generated.

Running the system simply consists in running the generated JAVA code on each machine, and interacting via interfaces to send messages from a process to another. Figure 5.14 shows the states of the frames presenting the automata after running the generated code and several interactions with the frames.

Figure 5.13: Enter the data about the servers

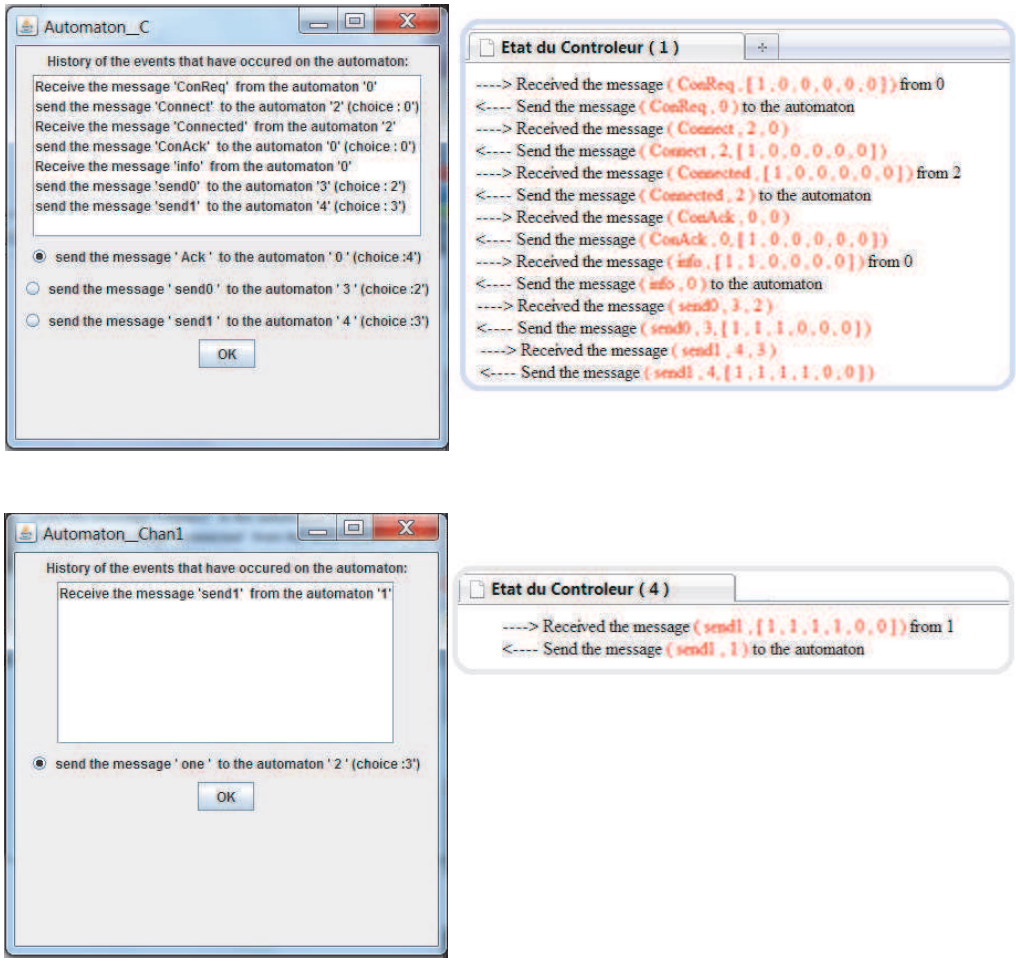


Figure 5.14: Example of a run showing the frame of the automaton A and its controller

A step-by-step execution of the Morse Code example is given in the appendix 7.3.2, and shows the evolution of the servers while running the execution presented in Figure 5.6.

5.6 localization of HMSC

We implemented the localization process presented in chapter 4. To illustrate the localization process we will use an example based on the toaster example from [53]. Figure 5.15 shows the graphical view of the HMSC and bMSCS of this example.

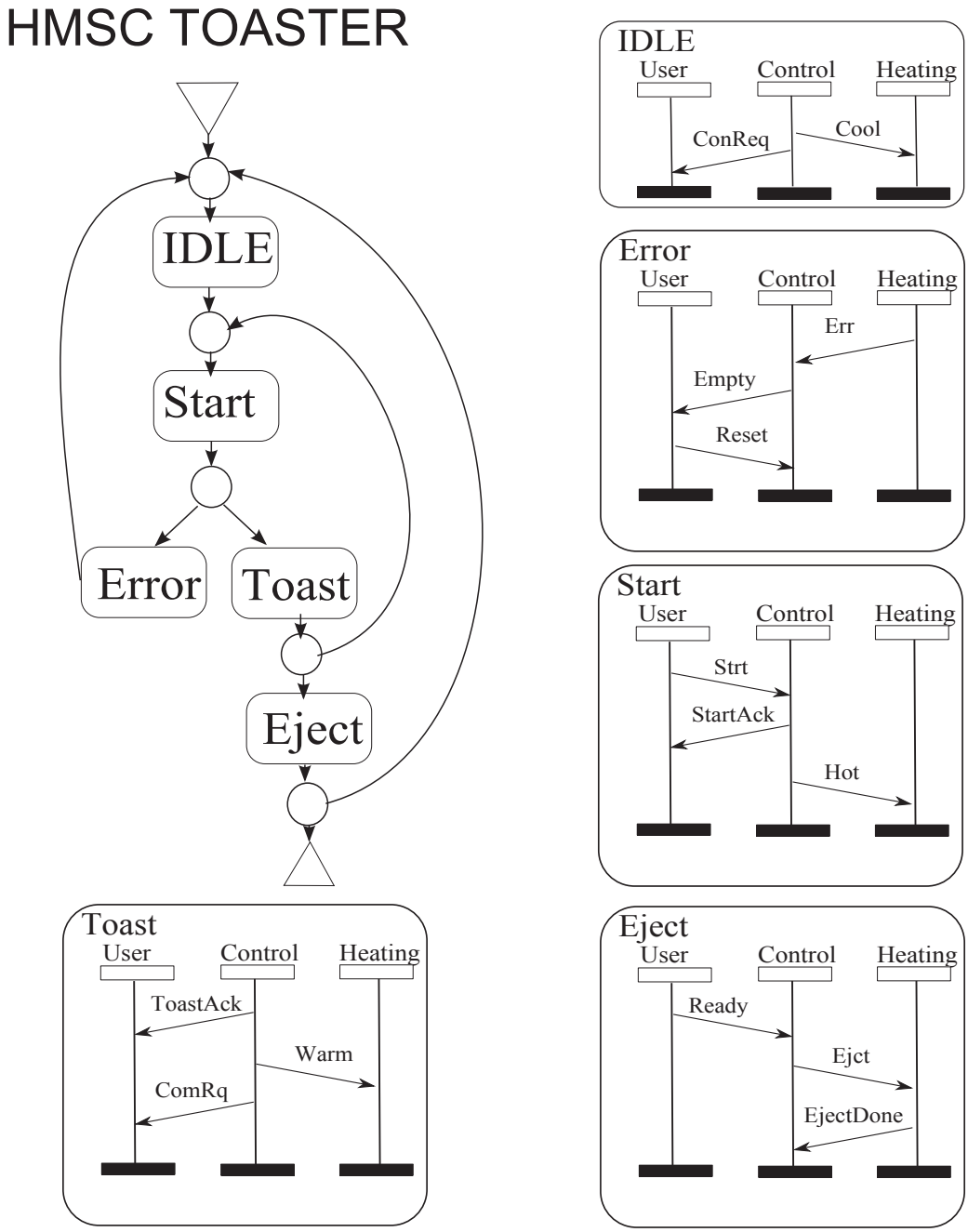


Figure 5.15: The toaster example

The specification consists of the HMSC Toaster which describes the composition of bMSCs IDLE, Eject, Error, Start, and Toast. Each of the bMSCs contains three processes User, Control and Heating. From a global perspective, the behavior of the toaster can be explained as follows. In IDLE the Control asks the User for a command

(ComReq) and advices Heating not to heat (Cool). Once the user decides to toast (Strt) the start is acknowledged (StartAck) and the Heating receives a command to start heating (Hot). In bMSC Toast, the beginning of the toasting process is reported to the user (ToastAck) and the heating is switched to keep the toast warm (Warm) and the user is asked by message (ComRq) to choose between another toasting period or ejecting the slice of bread. The Ready message will cause an Eject message (Ejct) to the heating which will respond with an EjectDone. The toaster returns to bMSC IDLE. When user asks for the toasting process to start, heating will check whether there is a slice of bread in the toaster or not. If not, it will send an Error message in bMSC Error and Control will tell the user that the toaster is empty (Empty). The Prolog code generated by SOFAT for the localization of this example can be found in the appendix 7.3.3.

This HMSC is clearly not local. We use SOFAT to generate the prolog code for the equivalent COP (Constraint Optimization Problem) of this HMSC. Then we execute this code using Sicstus prolog We got the solution in 16 ms (with the most performant heuristic between the tested ones "b/left/bisect/YXC", based on the statistical results presented in chapter 4). It consists simply in choosing the process Control as the leader in the bMSC Error. This can mean that for instance it is up to the control to check if there is no slice of bread and then it will signal it to the Heating process by sending a message. The solution proposed is not satisfactory. Note that in this case, the cost function is the one described in chapter 4, so it only cares about minimizing the number of messages. We can code another cost function that fits more with the desired result. On the other hand, this example shows the importance of the perspective that we have presented in chapter 4 that consists in adding constraints on the semantics. Then in the Toaster example adding constraints on the semantics may lead the solver to give an acceptable result.

Chapter 6

Conclusion and perspectives

6.1 Summary of contributions

In this thesis, we have studied the correct synthesis of *High-level Message Sequence Charts* (HMSC) models, that describe a global view of interactions in distributed system, into *Communicating Finite State Machines* (CFSMs) models that describe the behaviors for each process. We summarize our main contributions below:

- First we have considered the local HMSCs sub-class. The synthesis of CFSMs by a projection mechanism (the most intuitive way to implement an HMSC) is correct for a sub-class of local HMSCs, namely the reconstructible HMSCs. For local HMSCs that are not reconstructible projection mechanism may produce programs with more behaviors than in the HMSC specification. Then, we have proposed a solution to synthesize correct implementation for the local HMSCs that are not reconstructible: Additional controllers simply tag messages and delay them to ensure correct ordering of message receptions. These controllers need only a little information to ensure that the processes runs with respect to the specification: Each process executes its task as defined in the projection of the specification, and controllers ensure coordination. The results of Chapter 3 show that the projection of the behaviors of the controlled system on events of the original processes is equivalent (up to a renaming) to the behaviors of the original HMSC. One important aspect of this work is that processes and controllers are independent processing units, which communicate asynchronously and can be implemented on any distributed architecture. This provides a great

genericity for the method.

- Second, we have focused on solutions to synthesize CFSMs from non-local HMSCs. Indeed, a non-local HMSC can be transformed into a local HMSC by adding new synchronization messages. We have shown in Chapter 4 that this transformation can be automated as a constraint optimization problem. We allow additional active instances and new messages in bMSCs, but do not change the structure of the HMSC. The impact of modifications brought to the original specification can be minimized with respect to a cost function. This results in slight modifications of the original specification. The approach was evaluated on a large number of randomly generated HMSCs. The results of this experimentation show an average runtime of a few seconds, which demonstrates the applicability of the technique.

- We have implemented the previous approaches to allow the use of the algorithms in practice, and all these functionalities are added into an existing tool called SOFAT presented in Chapter 5. In this thesis, we have extended SOFAT with code synthesis functionalities, allowing generating communicating automata, Promela code, REST based web services from HMSCs, we have also implemented the localization procedure.

6.2 Future work

In the future, we envisage working in several different directions. First, we intend to continue to improve the algorithms and techniques that we have presented: Several perspectives were presented in Chapters 3 and 4 (like the integration of data and time, the reduction of tags, etc.) and for the localization procedure (the definition of new cost functions, add constraints on the semantics, etc.). These extensions are rather straightforward improvement of the techniques proposed in Chapter 3 and 4. We also plan to consider more involved extensions:

- consider how we can use HMSCs in a Software Product Line (SPL) context. SPL captures commonality and variability between a set of software products sharing a common, managed set of features that satisfy the specific needs of a particular market segment. Commonality designates the elements that are common to all products while variability designates the elements that may vary from a product to another one. For instance we might have a set of distributed systems S that

have the similar HMSCs specifications except for some specific bMSCs that vary from one system to another (this can be due to: the type of connection, type of machine that are used, etc.). In this case, the commonality designates all the bMSCs that are common to all the HMSCs of the systems and variability designates the bMSCs that varies from one HMSC to the other. The use of SPL in such cases aims at improving productivity and decreasing realization times by gathering the analysis, synthesis and implementation activities of all the set of distributed systems S.

- consider several types of communication channels (lossy, etc.).

Finally, we would like to highlight that our proposed synthesis work provides many important advantages that we can summarize in two points: First, the skeleton code generation, representing the interactions within the distributed system, helps the developers to avoid the problems caused by the concurrency in distributed systems. This eases, for instance, the generation of correct protocols. Furthermore, the use of optimization techniques in this synthesis results in more efficient protocols (minimizing the traffic over the network, etc.). Second, the controllers, required in such distributed environment, strengthen the advantages provided by the code synthesis approach. This is mainly due to their flexibility (there is no need to implement them on the same machines of their controlled entities) and the reduced amount of information required on the controlled entities.

Chapter 7

Appendix

7.1 Chapter 3

This section provides a proof for theorem 3.4.1, that is we want to show that the original specification given as a HMSC and the synthesized controlled machines exhibit the same behaviors. We proceed in several steps. We first show (lemma 7.1.1 that in the synthesized machines, all choices (i.e. events corresponding to the first event of some bMSC) are causally ordered in any execution. We then show (Lemma 7.1.2) that for every configuration of a HMSC H reachable after an execution, there exists a finite set of configurations of the synthesized machines reachable by observing the same execution. The last steps show inclusion of specification and implementations languages in both directions by contradiction. Supposing that one can reach a configuration (after executing a prefix O), where H allows firing of an event a but not corresponding configuration of the CFSM allow a leads to a contradiction for all types of events. We consider each type of events and show that the 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 of the synthesized language, or the fact that choices are ordered.
Let us first show that all choices in the synthesized machines are causally ordered.

Lemma 7.1.1. *For each local HMSC H, the choices events in any behavior of the synthesized communicating machines are totally ordered.*

Proof: We prove this property by induction. Let us denote by P_n the property: For all H, local HMSC, the choices in any behavior of the synthesized communicating

machines in a run containing n choices are totally ordered.
Let us first verify this property for $n = 2$. As H is local, then there is only one CFSM that can perform an action (a message sending) from the initial configuration. The next choice can then only be performed after the first one. Hence, the first two choices are ordered.
Let us suppose that the property is verified up to n, and prove that it also holds for $n + 1$. Let us suppose a prefix $O \circ O'$ from $\mathcal{L}(\| \ K(A_i)|C_i)$ with $n + 1$ choices, such that O contains n choices. Then, O is of the form $O = \{c_1\} \circ O_1 \ldots \{c_n\} \circ O_n$, where each c_i is a choice event, and such that $\{c_1\} \circ O_1$ is an execution of a prefix of the first bMSC M_1 appearing in this run. Then, $O \circ O'$ can be completed by piece P_1 such that $O \circ O' \circ P_1$ contains a complete execution of the first bMSC M_1 by the controlled CFSM (so far, nothing forces M_1 to be completely executed in $O \circ O'$).
We can now use a nice property of FIFO bMSCs: every bMSC M can be represented by one of its linearizations. Hence, knowing the respective ordering of actions on each process is sufficient to draw a bMSC. Let us now consider any bMSC of the form $M = P \circ P_a \circ P_b \circ P'$, where P, P' are pieces of bMSC, P_a and P_b are pieces containing only actions a and b, respectively. Then, if a and b are located on distinct instances, then M can also be written as $M = P \circ P_b \circ P_a \circ P'$. This property also applies to pieces of bMSCs, and also to CFSM executions, which can be seen as bMSC pieces.
In the behavior $O \circ O' \circ P_1$, all actions of P_1 are concurrent with actions from $\{c_2\} \ldots O_n \circ O'$, as otherwise at least one action in do not need to wait for the execution of an event in P_1 to be fireable, and $O \circ O'$ would not be an execution of our CFSM.
So, $O \circ O' \circ P_1$ can be equivalently rewritten as $O \circ O_{1,1} \circ \ldots O_{1,n} \circ P_1 \circ \{c_2\} \circ O'_2 \ldots \{c_n\} \circ O'_n \circ O'$, where each $O_{1,i}$ is the part of O_i that belongs to M_1 and $O'_i = O_i \setminus O_{1,i}$. Note that P_1 is ensured to be a legal continuation of $O \circ O'$ as no machine can start executing events with tags greater than $0^{\mathcal{B}_H}$ before executing all its tasks in M_1. This also means that one does not have to change the tag of messages appearing in $P1$ to rewrite $O \circ O' \circ P_1$ into $O \circ O_{1,1} \circ \ldots O_{1,n} \circ P_1 \circ \{c_2\} \circ O'_2 \ldots \{c_n\} \circ O'_n \circ O'$ (all messages between controllers in $P1$ will be tagged by a vector associating 0 to all branches except the branch labeled by $M1$ in H).
Let us denote by $P_{2,n} = \{c_2\} \ldots \{c_n\} \circ O'_n \circ O'$ the tagged piece starting at choice event c_2, and by $P'_{2,n}$ the same piece, where all tags are decremented on component M_1. $P'_{2,n}$ is a run with n choices of an HMSC H', which is a copy of H where the initial node is the node reached in H after M_1. Hence, all choices in $P'_{2,n}$ are ordered, and so are choices in w'. Hence, all choices in $O \circ O'$ are totally ordered. $\square$
As choice events are the only moment when a tag is updated, this lemma also means

that the set of tags that can appear in an execution is the set of tags labeling choice events, and hence that the tags produced in any run that belongs both to $\mathcal{L}(H)$ and $\mathcal{L}(\parallel K(A_i)|C_i)$ are the same. A *configuration* of an HMSC H is an element of $\mathcal{L}(H)$ (i.e. a bMSC piece). Note that each configuration in $\mathcal{L}(H)$ is a prefix of a bMSC generated by a unique minimal path ρ_P of H, as choice events uniquely designate chosen branches (this is ensured by our restrictions). We will say that an action a is fireable from a configuration P of H iff $P \circ \{a\} \in \mathcal{L}(H)$ (where $\{a\}$ is the bMSC piece that contains only action a. This means that either $P \circ \{a\}$ is a prefix of O_{ρ_P}, or that there exists a path $\rho' = \rho_P.(n, M, n')$ such that $P \circ \{a\}$ is a prefix of $O_{\rho'}$.
We can now show that for every prefix O that belongs to the language of H and to the language of the synthesized machines, one can find a finite sets of executions of the controlled architecture that are equivalent to O after renaming and erasing of controllers' events.

Lemma 7.1.2. *Let $O \in Ru(Unc(\mathcal{L}(\parallel K(A_i)|C_i))) \cap \mathcal{L}(H)$ be an execution. Then, there exists a finite set of executions $X = \{O_1, \ldots, O_k\}$ of $(\parallel K(A_i)|C_i)$ such that $Ru(Unc(X)) = \{O\}$.*

Proof: The events of the controlled automata in executions of the CFSM can be obtained from O by replacing every action on a process p by an action labeled by RU^{-1} in the CFSM execution (for instance $p!q(m)$ becomes $p!C_p(q, m, b)$ for some branch b). The behavior on each controller simply consists in receiving messages from the automaton it controls, and forwarding them to the next controller, or conversely receiving messages from a controller and forwarding them to the automaton they control in the order specified by the branches. Then, every complete message from p to q in O can be mapped to a sequence of 3 messages that "simulate" the sending from a process p to a process q. So, if O has no unreceived message, then all automata in $(\parallel K(A_i)|C_i)$ are in a configuration with empty communication buffers, and each automaton and controller can only be in one state. Now if there is at least one message m sent from p to q in O but not received, then this means that $(\parallel K(A_i)|C_i)$ is in a configuration where a message (q, m, b) can be transiting between p and C_p, a message (m, τ) can be transiting between C_p and C_q, or last a message (p, m, b) can be transiting between C_q and q. Figure 7.1-a) shows an execution of some HMSC in which a message $m3$ is sent but not yet received. There can be three configurations corresponding to such situation, and Figure 7.1-b) shows one of them in which a message of type $m3$ is transiting between the controllers of B and A. Hence, the number of configurations in which CFSMs can be while observing $O \in Ru(Unc(\mathcal{L}(\parallel K(A_i)|C_i))) \cap \mathcal{L}(H)$ and the size of X are finite and depend on the number of unreceived messages. □

From this lemma, one can also deduce that there exists a correspondence between each configuration reachable in the semantics of H and a finite set of configurations of the synthesized machines.

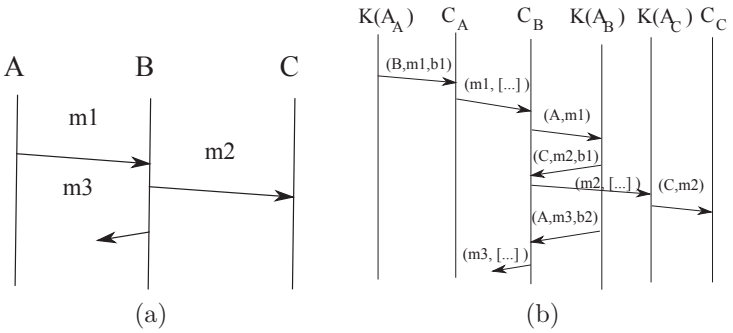


Figure 7.1: Relating HMSCs executions and CFSMs

We are now ready to prove language equality, by showing two inclusions.

Lemma 7.1.3. *Let H be a HMSC, $\{A_i\}_{i\in I}$ and $\{A_i\}_{i\in I}$ be respectively the projection of H on its instances, and the synthesized controllers. Then, $Ru(Unc(\mathcal{L}(\| K(A_i)|C_i))) \subseteq \mathcal{L}(H)$*

Proof: For short, we write $\mathcal{L}_1 \subseteq \mathcal{L}_2$ instead of $Ru(Unc(\mathcal{L}(\| K(A_i)|C_i))) \subseteq \mathcal{L}(H)$.
Suppose that there exists a prefix $O \circ \{a\} \in \mathcal{L}_1$ such that $O \in \mathcal{L}_2$, but $O \circ \{a\} \notin \mathcal{L}_2$. O is a configuration of H, and as $O \in \mathcal{L}_1$, there exists a set X_O of possible executions of the synthesized CFSM such that $RU(Unc(X_O))$ (from lemma 7.1.2). There also exists at least one execution $O_i \in X_O$ such that after executing O_i, the CFSM is in a configuration C_A in which automaton $K(A_p)$ is in a state allowing firing of a transition (s, σ, s') with $Ru(\sigma) = a$.
Suppose that a is a sending event from p to q, i.e. $a = p!q(m)$ for some m, and $\sigma = K(a_p)!C_p(m, b)$ for some branch b. The sequence of events in O on p and in O_i on $K(A_p)$ are identical, up to renaming. Hence, this means that p and $K(A_p)$ follow the same path ρ of H until the end of O_i (recall that transitions of projected automata are defined from transitions of H). Then, all predecessors of σ in O_i allowing to reach state s have been executed, and all predecessors of a in O_ρ have been executed too in O. Hence, O is a configuration of H that allows for the firing of action a on process p, as projection preserves (up to renaming due to control) sequences of events on each process. This contradicts the fact that a is a sending event. A similar case holds for atomic actions. Hence, a can only be a receive action, i.e. a is of the form $a = p?q(m)$ for some q, m, and $\sigma = K(A_p)?C_p(q, m)$. This means that O_i is an execution that

brings the CFSM in a configuration in which the FIFO queue from C_p to $K(A_p)$ has a message m as head (otherwise (q, σ, q') can not be fired).
As mentioned in lemma 7.1.2, messages in $Ru(Unc(O_i))$ are simulated by three messages in O_i. Then, O_i is of the form $O_1 \circ \{\sigma_1\} \circ \{\sigma_2\} \circ \{\sigma_3\} \circ \{\sigma_4\} \circ \{\sigma_5\} \circ O_2$, where O_1 is a prefix and O_2 is a piece. We furthermore have $\sigma_1 = K(A_q)!C_q(p, m, b)$ for some branch b, $\sigma_2 = C_q?K(A_q)(p, m, b)$,$\sigma_3 = C_q!C_p(m, \tau)$, $\sigma_4 = C_p?C_q(m, \tau)$, and $\sigma_5 = C_p!K(A_p)(q, m, b)$. If any of these actions is missing in O_i, then σ can not be fired. Such situation is depicted in Figure 7.3-a).
Let us now consider O as a configuration of H. There is a message m sent from q to p but not yet received in O. Event a is not allowed by H from configuration O, however, message m was sent. Hence, a is forbidden because according to the chosen path in H, there are some events $\alpha_1, \ldots \alpha_k$ to execute on instance p before a (i.e. there is piece of bMSC P_a such that $O \circ \{a\}$ is not a configuration of H, but $O \circ P_a \circ \{a\}$ is). This situation is depicted in Figure 7.3-b).
After execution O_i, the automaton $K(A_p)$ has reached a state s, which means that s is reachable in $K(A_p)$ by reading the controlled version of the actions appearing on p in O (i.e. the sequence $Ru^{-1}(\pi_p(O))$). As there exists a transition by (s, σ, s') in $K(A_p)$ and as we know that $\alpha_1, \ldots \alpha_k$ can be executed by process p after O, then state s is a choice, from which at least two transitions (s, σ, s') and (s, c, s_1), where c is an action of the automata corresponding to α_1 (i.e. $RU(c) = \alpha_1$), can be fired. Note that as all choices in H are local, c is necessarily a message reception event.
Events c and σ belong to different branches of the same choice of H, and we have that $\tau(c) \neq \tau(a)$, as events of P_a located on p **have to** be executed before a. From lemma 7.1.1 we know that all choices in an execution of the CFSM are totally ordered. Furthermore the tags associated to an execution of an HMSC and to an execution of the synthesized communicating automata are the same. We then have $\tau(c) < \tau(a)$. As c and a are events of choices that concern p, the communication $\sigma_4 = C_q!C_p(m, \tau = \tau(a), b)$ that must occur in O_i before σ can not be executed by $K(A_p)$ as the message received by C_p at event σ_4 is tagged by a vector τ which is not the expected successor tag on C_p. Hence C_p can not consume it and forward m to $K(A_p)$, unless it has received and forwarded the messages corresponding to the second branch of H, which does not appear in O. Then, receptions on this branch must be executed by $K(A_p)$ before σ. We then have a contradiction, and $\mathcal{L}_1 \subseteq \mathcal{L}_2$. $\square$

Lemma 7.1.4. *Let H be a HMSC, $\{A_i\}_{i \in I}$ and $\{A_i\}_{i \in I}$ be respectively the projection of H on its instances, and the synthesized controllers. Then, $\mathcal{L}(H) \subseteq Ru(Unc(\mathcal{L}(\| K(A_i)|C_i)))$*

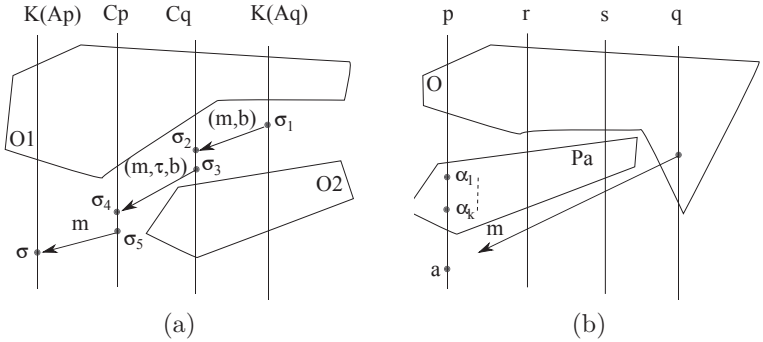


Figure 7.2: Illustration of the proof of Lemma 7.1.3

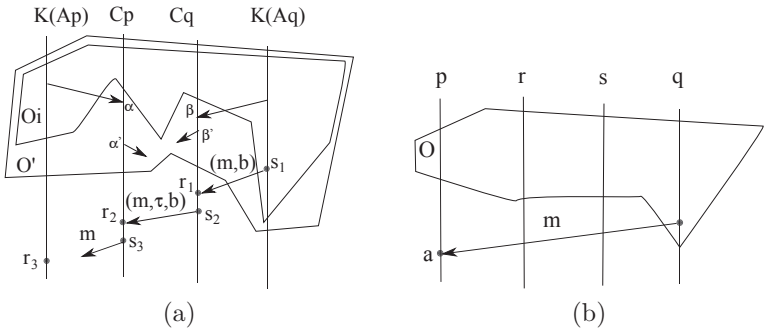


Figure 7.3: Illustration of the proof of Lemma 7.1.4

Proof: For short, we write $\mathcal{L}_2 \subseteq \mathcal{L}_1$ instead of $\mathcal{L}(H) \subseteq Ru(Unc(\mathcal{L}(\| K(A_i)|C_i)))$.
Let us suppose there exists $O \circ \{a\}$ such that $O \circ \{a\} \in \mathcal{L}_2$, $O \in \mathcal{L}_1$ but $O \circ \{a\} \notin \mathcal{L}_1$. From lemma 7.1.2, there exists an execution O_i of the CFSM such that $Ru(Unc(O_i)) = O$. If a is a sending of a message or an atomic action on process p, then $K(A_p)$ must be in a state q from which an transition (q, σ, q') with $RU(\sigma) = a$ is fireable, as the sequence of controlled events corresponding to the projection of O on p is recognized by $K(A_p)$, and as $K(A_p)$ is a deterministic machine. Transition (s, σ, s') is fireable as soon as $K(A_p)$ is in state s, which is the case, and $O_i \circ \{\sigma\}$ is a behavior of the CFSM. So, if a is a sending event or an atomic action, $O \circ \{a\} \in \mathcal{L}_1$, which contradicts the initial hypothesis.
Then, a is a reception $a = p?q(m)$, an O looks like the execution represented in Figure 7.3-d). As shown in lemma 7.1.2, a message m from p to q in a configuration of H corresponds to a sequence of three messages in the CFSM execution: $s_1 = K(A_q)!C_q(p, m, b)$, $r_1 = C_q?K(A_q)(p, m, b)$, $s_2 = C_q!C_p(m, \tau)$, $r_2 = C_p?C_q(m, \tau)$, $s_3 = C_p!K(A_p)(q, m, b)$, $r_3 = K(A_p)?C_p(q, m, b)$ (with $Ru(Unc(r_3 = a))$). As the sending of m from q to p appears in execution O, the corresponding sending event s_1 executed by $K(A_q)$ also appears in O_i.
We can now proceed as follows: we first prove that there exists an execution O' of the synthesized CFSM such that $Ru(Unc(O')) = O$, and such that in the configuration reached by the CFSM after O', controller C_p is ready to execute event r_2 if a message is buffered with appropriate tag. We then show that such message exists, and that it is correctly tagged, and hence allows for the reception of r_2, followed by s_3 and r_3.
Execution O is a prefix of a concatenation of bMSCs labeling branches of H, i.e. a sequence of bMSCs $M_1 \circ M_2 \circ \dots M_n$. Among these bMSCs, process p is concerned only by a subset $M_{i1}, \dots M_{ik}$ of them, and process q by another subset $M_{j1}, \dots M_{jk'}$. Sending and reception of message m is O belongs to a bMSC M_{pq} appearing in both sets. From Lemma 7.1.2, as $O \in \mathcal{L}_1 \cap \mathcal{L}_2$, there exists an execution O_i of the CFSM such that $Ru(Unc(O_i)) = O$. After O_i, the CFSM is in a configuration in which automaton $K(A_p)$ can fire a transition (s, r_3, s') provided the head of the queue from C_p to $K(A_p)$ is a message m.
One can note that the controller C_k of an automaton $K(A_k)$ systematically receives messages sent by $K(A_k)$ (rule R1) and forwards them to another controller. Similarly, if C_k receives a message from another controller, it forwards it to $K(A_k)$. This means that when A_k receives a message and this reception belongs to a branch b of H, then C_k has necessarily counted this branch in its vectorial clock τ_k, that remembers the number of occurrences of choices concerning k that have occurred so far. This also means that C_k has accepted an incoming message (m, τ) coming from another

controller, and that τ was a valid tag at the time of this message reception.
Let us consider again O_i. This execution is a partial execution of $M_1 \circ \dots M_n$ by the CFSM, and contains some elements of M_{pq}, including event s_1. Considering the sequence of events executed by $K(A_p)$ and $K(A_q)$ in O_i, one can also get the sequence of sendings/receptions executed by the controllers C_p and C_q, as for every event of the form $p!q(m)$ in O there exists a pair of events $K(A_p)!C_p(m,b).C_p?K(A_p)(m,b)$, and for every event of the form $p?q(m)$ in O there exists a pair of events $C_p!K(A_p)(i,m,b).K(A_p)?C_p(i,m,b)$ in O_i. However, this does not mean that C_q is ready to execute(or has already executed) r_1 or C_p is ready to execute (or has already executed)r_2 , as some messages may still need to be consumed in the message queues of C_p and C_q. Let us suppose that C_q must receive at least one message, either from another controller, or from $K(A_q)$ before receiving message m. Let us call this reception β and the following retransmission β'. In the first case, β **must** be executed before r_1 if and only if it is a reception of a message that have to be executed to comply with the sequence of receptions defined in some branch of H. In this case, β' is a sending of a message to $K(A_p)$, and as it has to be executed before r_1, then it means that some reception on $K(A_q)$ must be executed before s_1, and then we cannot have $Ru(Unc(O_i)) = O \in \mathcal{L}_1 \cap \mathcal{L}_2$. In the latter case, as reception of messages from controlled automata can be performed without waiting (according to rule R1 of the controllers), then there exists an execution $O_i \circ \{\beta\} \circ \{\beta'\}$ of the CFSM from which r_1 can be executed, and such that $Ru(Unc(O_i \circ \{\beta\} \circ \{\beta'\})) = O$. Similarly, if C_p is in a configuration from which r_3 can not be fired because a reception α followed by a retransmission of message α' must occur before r_3, then one can show that either this implies that $Ru(Unc(O_i)) \notin \mathcal{L}_1 \cap \mathcal{L}_2$, or that there exists an execution $O_i \circ \{\alpha\} \circ \{\alpha'\}$ such that $Ru(Unc(O_i \circ \{\alpha\} \circ \{\alpha'\})) = O$. Figure 7.3-c) illustrates this situation. The argumentation extends for arbitrary sequences of actions $w_p = \alpha_1.\alpha'_1 \dots \alpha_i.\alpha'_i$ and $w_q = \beta_1.\beta'_1 \dots \beta_j.\beta'_j$, $i, j \in \mathbb{N}$ that have to be executed by C_p and C_q before the execution of r_1 and r_2: mandatory reception from a controller implies $Ru(Unc(O_i)) \notin \mathcal{L}_1 \cap \mathcal{L}_2$, and mandatory reception from $K(A_p)$ or $K(A_q)$ can be performed to obtain a larger execution. Note that in the sequences of missing events w_p and w_q C_p and C_q can not be forced to exchange a message, which would imply a reception from another controller, and hence $Ru(Unc(O_i)) \notin \mathcal{L}_1 \cap \mathcal{L}_2$. So events in w_p are independent from events in w_q, and one can find an execution of the CFSM O' that includes O_i, the sequences w_p, w_q, and the two events r_1 and s_2. Hence, after O', controller C_p is in a configuration allowing it to receive the message (m, τ) sent by C_q if τ is a correct tag. This reception corresponds to rule (R2) of the controller. If r_2 can not be executed by C_p but $[\tau_p]_p = [\tau]_p$, then it usually means that r_2 is not the next reception to perform according to the chosen branch, and

that there are remaining actions to perform on C_p before allowing r_2. However, we have ruled out this possibility after execution of O'. Hence, the only case remaining is when $[\tau_p]_p \neq [\tau]_p$, and $[\tau]_p$ is not an immediate successor of $[\tau_p]_p$. However, we know that s_1 is a causal consequence of all choices that have been performed in O' up to bMSC M_{pq}, as one can establish a correspondence between messages in O and sequences of messages in O'. So, $\tau[b]$ is exactly the number of occurrences of branch b in $M_1 \circ \cdots \circ M_{pq}$. As C_p has executed all events in w_p required before execution of r_2, that is corresponding to events in $M_1 \circ \cdots \circ M_{pq-1}$ in execution O', and more precisely all receptions of messages coming from other controllers, we necessarily have $[\tau_p]_p[b'] = [\tau]_p[b']$ for every branch $b' \neq b$ of H, and $[\tau_p]_p[b] + 1 = [\tau]_p[b]$. This contradicts the fact that r_2, necessarily followed by s_3 and r_3 can not be executed from O', and hence contradicts $O \circ \{a\} \notin \mathcal{L}_1$. $\square$

Theorem 7.1.4. *Let H be an HMSC, and let $\|_{i \in I} K(A_i)|C_i$ be its controlled synthesis. Then, $Ru(Unc(\mathcal{L}(\|_{i \in I} K(A_i)|C_i))) = \mathcal{L}(H)$.*

Proof: The proof of this theorem is is straightforward, as we have inclusion of languages in both directions (lemmas 7.1.3 and 7.1.4).

7.2 Chapter 4

7.2.1 Proof of correctness of theorem 4.4.1

Theorem 7.2.1. *Computing solutions for a localization problem using an optimal solution search for the corresponding constraint model is both a sound and complete algorithm.*

To establish soundness and correctness, we first need a technical lemma establishing correspondence between solutions of CP_H and localized HMSCs.

Lemma 7.2.1. *Let H be a HMSC, and CP_H be the associated constraint problem. For every (not necessarily minimal) solution s of CP_H of cost $\mathcal{F}(s)$, there exists a localized extension H_s of H such that $\mathcal{F}(H_s) = \mathcal{F}(s)$.*

Proof : Obviously, for every solution $s = \{X_i, Y_i\}_{i \in 1..|\mathcal{M}|}$, there exists a localized extension H_s such that for every $M_i \in \mathcal{M}$, $f(M_i)$ has X_i as instance set, and Y_i as leader instance. The cost $\mathcal{F}(H_s)$ of any localized extension H_s is $\sum c(M_i, f(M_i))$. The value $\sum c(M_i, f(M_i))$ be achieved by designing each M'_i as follows: set $M'_i = prefix_i \circ M_i$, where $prefix_i$ is:

- a bMSC containing messages from Y_i to any instance in $\Big(\phi(min(M_i)) \setminus \{Y_i\}\Big) \cup \Big(\phi(M'_i) \setminus \phi(M_i)\Big)$ if $Y_i \in \phi(min(M_i))$
- a bMSC containing messages from Y_i to any instance in $\phi(min(M_i)) \cup \Big(\phi(M'_i) \setminus \phi(M_i)\Big) \setminus \{Y_i\}$ if $Y_i \in \Big(\phi((M'_i)) \setminus \phi(M_i)\Big)$
- a bMSC containing messages from Y_i to any instance in $\phi(min(M_i)) \cup (\phi(M'_i) \setminus \phi(M_i))$ if $Y_i \in \phi(M_i) \setminus \phi(min(M_i))$

Obviously, taking as leaders and instance sets the choices indicated by solution s to CP_H, and designing $\mathcal{M}' = \{M'_i\}_{i \in 1..|\mathcal{M}|}$ as defined above, we necessarily have that H_s is localized: all bMSCs are localized, and equality constraints impose that two transitions originating from the same node are labeled by bMSCs with the same leaders. Inclusion constraints has as a consequence that for every path $\rho = (n, M'_1, n_1) \dots (n_{k-1}, M'_k, n_k)$, $\phi(min(M^\rho)) = \phi(min(M'_1))$. $\square$

Proof (of Theorem 4.3.1): We can now proceed in two directions, showing first soundness, i.e for every optimal solution $\{X_i, Y_i\}$ to a constraint problem CP_H, there exists a localized extension H' which is minimal w.r.t the cost function $\mathcal{F}$. Let s be an optimal solution. By lemma 7.2.1, we know that there exists a localized extension H_s with the same cost as s. Now, suppose that there exists H' a localized extension of H, such that $\mathcal{F}(H') < \mathcal{F}(H_s)$. Let f' be the relation mapping bMSCs of $\mathcal{M}$ to bMSCs of $\mathcal{M}'$. Then there are several bMSCs $M_{i_1}, \dots, M_{i_k}$ such that $c(M_i, f'(M_i)) < c(M_i, f(M_i))$, i.e. they are defined over sets of instances $X_{i_1}, \dots, X_{i_k}$ with leaders $Y_{i_1}, \dots, Y_{i_k}$, and still satisfy equality and inclusion constraints. Hence, the solution s cannot be optimal as s' obtained by replacing each X_i (resp Y_i) by $\phi(f'(M_i))$ (resp $\phi(min(f'(M_i)))$ is better than s. Contradiction.
Let us now prove completeness, that is that every optimal localized extension H' of H is defined over a set of bMSCs $\{M'_i = f(M_i)\}_{i \in 1..|\mathcal{M}|}$ is such that $s = \{(\phi(M'_i), \phi(min(M'_I)))\}_{i \in 1..|\mathcal{M}|}$ is also an optimal solution for CP_H. Obviously, as H' is localized, s satisfies all equality and inclusion constraints imposed by CP_H, otherwise one can find a path ρ in H with $|\phi(min(M^\rho))| > 1$, or two paths with distinct minimal instances. Now, let us suppose that s is not optimal, that is, there exists $s' = \{(X_i, Y_i)\}_{i \in 1..|\mathcal{M}|}$ such that $\mathcal{F}(s') < \mathcal{F}(s)$. Using lemma 7.2.1, we know that there exists a localized extension $H_{s'}$ with the same cost $\mathcal{F}(H_{s'}) = \mathcal{F}(s')$. Hence H' is not optimal, contradiction. $\square$

7.3 Chapter 5

7.3.1 Promela code generated for the Morse Code example

Now, let us explain the generated Promela code of the Morse Example in details: The Promela program instantiates a process for each instance of the HMSC at system-setup time. This is implemented in the *init* section in Promela. Messages in Promela can be typed. We choose *mtype* to construct the message types. To model the message exchanges, we use channels with a given capacity (this is mandatory in Promela). Note that in HMSCs, a bound on the maximal number of messages sent and not yet received does not necessarily exists. Such a maximal bound exists when the considered HMSC is not *divergent* [15]. In practice, imposing a bound on channels is a good way to prototype a protocol, and get extensive analysis of the behaviors of a protocol "up to some bound". In Promela, implementation of an HMSC has the following overall syntactic structure [52]:

- the Promela code first gives the necessary data definitions, including the global channels declarations denoted by the keyword *chan*.
- Next, the definition of the process bodies indicated by the keyword *proctype*.
- Finally, the instantiation of the whole system using an *init* statement.

In the example below, we define three sets of channels. The set denoted by Aut_Cont represents the channels going from the automaton to its controller, and the set $Cont\_Aut$ represents the channels going from a controller to its automaton. Hence channel $Aut\_Cont[i]$ denotes the channel going from the process P_i to its controller C_i. The set of channels denoted by P is a matrix of channels connecting all the pairs of controllers. For example, the channel $P[i].PP[j]$ is the channel carrying messages from the controller of the process P_i to the controller of the process P_j.
The *Mask* structure represents the existence of a process in a choice. It is used filter branches of a choice vector which do not concern a particular process. For instance $Mask[0].T[5] = 1$ means that the process P_0 exists in the choice (or bMSC) $M5$, and the statement $Mask[3].T[1] = 0$ means that the process P_3 does not exist in the choice $M1$. We do not write this last statement in the code because in Promela all the variables are set to zero by default.
The structure defined by $PI[NBC].T[AUTNUM]$ represents the number of events that a process should do in a branch. For instance $PI[0].T[0] = 2$ means that the process P_0 (A) is executing two actions in the branch $M0$ (these actions are the

sending of the message *conReq* and the reception of the message *conAck*). The structure $PIC[NBC].theAUT[AUTNUM].recpt[MAXEVTNUM]$ indicates the order in which receptions should occur within a bMSC.
For instance, $PIC[0].theAUT[2].recpt[0] = 0$ means that in the choice $M0$ the process P_2 (C) should first receive a message coming from the process P_0 (A), and $PIC[0].theAUT[2].recpt[1] = 1$ means that the second reception that the process P_2 should do is from the process P_1 (B).
Executing a statement of the form $xy?b$ means that the reception of a message of type b via the channel xy is performed (consumed). The other form is $xy? < b >$ is just to test if the message b is present at the head of the channel. It is not consumed. This construct is used to test the received tag. The control of the order of receptions is done by an active waiting.
Similarly, the statement $wz!t$ means the sending of a message of type t via the channel wz. We define two types of exchanged messages $\{byte, mtype, byte\}$ and $\{mtype, tagtype\}$ respectively for the messages exchanged between automata and their controller, and another one for the exchanges among controllers.
The construct $\rightarrow$ serves as an enabling operator such that the operation on its right is only enabled if the guard on its left is *true*. The macro *next* takes the value *true* if the projections of tag of the received message on the choices where the controlled process exists is the direct successor of the local tag on the controller that receives the message. Similarly the macro *same* is *true* if the projections of the received tag and the local tag are the same. The macro $diff$ returns the value of the new choice that is currently executed.
We will explain the code of the process A, and then it will be the same for the code of the other processes. The $proctype\, Cont(int\, i)$ represents the controller of the process i. It implements the generic controller algorithm previously shown in the paper. At the end of the code, instruction $run\, Cont(3)$, for instance, creates the controller for process 3 (the process *chan*0). Each controller have its local variables: t is used to get the tag of the received message over channels communicating with other controllers. *tau* is used for the local tag. *nbevt* is set at each time we have a new choice to be executed. It helps the controller to know how many messages stamped with the current tag it should receive before allowing the reception of messages tagged with the direct successor tag. *Rec* is used to tell the controller in which order it should receive these events tagged with the same tag. Once a reception is done the controller shifts to the next channel on which it should receive a message by using the macro $Shift\_Rec$. Variable j is to perform a round-robin test of messages on different channels.

In the *init* section we initialize the values of the different structures and we create the processes. Note that the global variables (like NBC and AUTNUM) contain some static information about the HMSC that have been extracted during the compilation phase. They are used by the processes but do not serve as a communication mean.

```
/* Promela generated code of a CFM generated from an HMSC entry */
#define CMAX 5 /* max size of channels */
#define NBC 6 /* number of MSCs (choices) */
#define AUTNUM 5 /* the number of automata */
#define next ((t.T[0]-tau.T[0])*Mask[i].T[0]+t.T[1]-tau.T[1])
   *Mask[i].T[1]+t.T[2]-tau.T[2])*Mask[i].T[2]+t.T[3]-tau.T[3])
   *Mask[i].T[3]+t.T[4]-tau.T[4])*Mask[i].T[4]+t.T[5]-tau.T[5])
   *Mask[i].T[5])==1)
#define same ((t.T[0]-tau.T[0])*Mask[i].T[0]+t.T[1]-tau.T[1])
   *Mask[i].T[1]+t.T[2]-tau.T[2])*Mask[i].T[2]+t.T[3]-tau.T[3])
   *Mask[i].T[3]+t.T[4]-tau.T[4])*Mask[i].T[4]+t.T[5]-tau.T[5])
   *Mask[i].T[5])==0)
#define update tau.T[0]=t.T[0];tau.T[1]=t.T[1];tau.T[2]=t.T[2];
tau.T[3]=t.T[3];tau.T[4]=t.T[4];tau.T[5]=t.T[5]
#define diff (t.T[0]-tau.T[0])*Mask[i].T[0]*0+(t.T[1]-tau.T[1])
   *Mask[i].T[1]*1+(t.T[2]-tau.T[2])*Mask[i].T[2]*2+(t.T[3]-tau.T[3])
   *Mask[i].T[3]*3+(t.T[4]-tau.T[4])*Mask[i].T[4]*4+(t.T[5]-tau.T[5])
   *Mask[i].T[5]*5
#define MAXEVTNUM 2/*the max numb of rec in a bMSC*/
#define Shift_rec Rec.recpt[0]=Rec.recpt[1]; Rec.recpt[1]=-1
typedef tagtype { byte T[NBC]; }
typedef com_Num { byte T[AUTNUM]; }
com_Num PI[NBC] /* number of com events in a bMSC */
typedef order_recpt{byte recpt[MAXEVTNUM] } /* required seq */
typedef aut_choice{order_recpt theAUT[AUTNUM] }
aut_choice PIC[NBC]
typedef chan_col { chan PP[AUTNUM]=[CMAX] of {mtype,tagtype};}
chan_col P [AUTNUM];
chan Aut_Cont[AUTNUM]=[CMAX] of {byte,mtype,byte};
chan Cont_Aut[AUTNUM]=[CMAX] of {byte,mtype};
mtype = {ConReq, Connect, ConAck, Connected, info, send0, zero,
 send1, one, Ack, disConReq, disconnect, disConAck, disconnected }
 /*messages types in the system */
```

```
tagtype Mask[AUTNUM]; /* selection of concerned instances */

proctype A
{
sA0  : if
:: Aut_Cont[0]!C,ConReq,0; goto sA1
fi;
sA1  : if
:: Cont_Aut[0]?C,ConAck; goto sA2
fi;
sA2  : if
:: Aut_Cont[0]!C,info,1; goto sA3
fi;
sA3  : if
:: Cont_Aut[0]?C,Ack; goto sA4
fi;
sA4  : if
:: Aut_Cont[0]!C,disConReq,5; goto sA5
:: Aut_Cont[0]!C,info,1; goto sA3
fi;
sA5  : if
:: Cont_Aut[0]?C,disConAck; goto sA0
fi;
}

 proctype B()
{
sb0: if
 :: Cont_Aut[1]?2,Connect;goto sb1
 fi;
sb1: if
 :: Aut_Cont[1]!2,Connected,-1; goto sb2
 fi;
sb2: if
 :: Cont_Aut[1]?3,zero;   goto sb2
 :: Cont_Aut[1]?4,one;  goto sb2
 :: Cont_Aut[1]?2,disconnect;  goto sb3
 fi;
```

```
sb3: if
 :: Aut_Cont[1]!2,disconnected,-1;  goto sb0
 fi;
}

 proctype C()
{
sc0: if
 :: Cont_Aut[2]?0,ConReq;goto sc1
 fi;
sc1: if
 :: Aut_Cont[2]!1,Connect,-1;goto sc2
 fi;
sc2: if
 :: Cont_Aut[2]?1,Connected;goto sc3
 fi;
sc3: if
 :: Aut_Cont[2]!0,ConAck,-1;goto sc4
 fi;
sc4: if
 :: Cont_Aut[2]?0,info;goto sc5
 fi;
sc5: if
 :: Aut_Cont[2]!4,send1,3; goto sc6
 :: Aut_Cont[2]!3,send0,2;goto sc6
 fi;
sc6: if
 :: Aut_Cont[2]!4,send1,3; goto sc6
 :: Aut_Cont[2]!3,send0,2;goto sc6
 :: Aut_Cont[2]!0,Ack,4;goto sc7
 fi;
sc7: if
 :: Cont_Aut[2]?0,info;goto sc5
:: Cont_Aut[2]?0,disConReq;goto sc8
 fi;
sc8: if
 :: Aut_Cont[2]!1,disconnect,-1;goto sc9
```

```
 fi;
sc9: if
 :: Cont_Aut[2]?1,disconnected;goto sc10
 fi;
sc10: if
 :: Aut_Cont[2]!0,disConAck,-1;goto sc0
 fi;
}

 proctype chan0()
{
schan00: if
 :: Cont_Aut[3]?2,send0;goto schan01
 fi;
schan01: if
 :: Aut_Cont[3]!1,zero,-1;goto schan00
 fi;
}

 proctype chan1()
{
schan10: if
 :: Cont_Aut[4]?2,send1;goto schan11
 fi;
schan11: if
 :: Aut_Cont[4]!1,one,-1;goto schan10
 fi;
}

proctype Cont(int i) /* The generic controller */
{tagtype tau,t; byte nbevt, currentb; byte j=0; byte b;
  mtype m; order_recpt Rec;
do
/* RULE 1 */ :: Aut_Cont[i]?<j,m,b>;Aut_Cont[i]?j,m,b ->
   if  :: (nbevt==0) -> tau.T[b]++; nbevt=PI[b].T[i]-1;
   Rec.recpt[0]=PIC[b].theAUT[i].recpt[0];
   Rec.recpt[1]=PIC[b].theAUT[i].recpt[1];
```

```
   P[j].PP[i]!m,tau;
:: else -> nbevt--; P[j].PP[i]!m,tau
fi;
/* RULE 2 */ :: P[i].PP[j]?<m,t> ->
 if :: (same && (  Rec.recpt[0]==j)); P[i].PP[j]?m,t ->
  nbevt--; Cont_Aut[i]!j,m ; Shift_Rec;
:: else -> if  :: ((nbevt==0) && next); P[i].PP[j]?m,t ->
currentb=diff; update;nbevt=PI[currentb].T[i]-1;
Rec.recpt[0]=PIC[currentb].theAUT[i].recpt[1];
Cont_Aut[i]!j,m
::else->skip
fi;
 fi;
 :: j=(j+1)%AUTNUM;
od
}

init{ /* Constant values obtained from the HMSC parsing */

PI[0].T[0]=2;PI[0].T[1]=4;PI[0].T[2]=2;PI[1].T[0]=1;
PI[1].T[1]=1;PI[2].T[0]=1;PI[2].T[1]=2;PI[2].T[2]=1;
PI[3].T[0]=1;PI[3].T[1]=2;PI[3].T[2]=1;PI[4].T[0]=1;
PI[4].T[1]=1;PI[5].T[0]=2;PI[5].T[1]=4;PI[5].T[2]=2;

PIC[0].theAUT[0].recpt[0]=1;PIC[0].theAUT[0].recpt[0]=-1;
PIC[0].theAUT[1].recpt[0]=0;PIC[0].theAUT[1].recpt[1]=2;
PIC[0].theAUT[2].recpt[0]=1;PIC[0].theAUT[2].recpt[0]=-1;

PIC[1].theAUT[0].recpt[0]=-1;PIC[1].theAUT[0].recpt[1]=-1;
PIC[1].theAUT[1].recpt[0]=0;PIC[1].theAUT[1].recpt[0]=-1;

PIC[2].theAUT[0].recpt[0]=-1;PIC[2].theAUT[0].recpt[1]=-1;
PIC[2].theAUT[1].recpt[0]=1;PIC[2].theAUT[1].recpt[0]=-1;
PIC[2].theAUT[2].recpt[0]=3;PIC[2].theAUT[2].recpt[0]=-1;

PIC[3].theAUT[0].recpt[0]=-1;PIC[3].theAUT[0].recpt[1]=-1;
PIC[3].theAUT[1].recpt[0]=1;PIC[3].theAUT[1].recpt[0]=-1;
```

```
PIC[3].theAUT[2].recpt[0]=4;PIC[3].theAUT[2].recpt[0]=-1;

PIC[4].theAUT[0].recpt[0]=1;PIC[4].theAUT[0].recpt[0]=-1;
PIC[4].theAUT[1].recpt[0]=-1;PIC[4].theAUT[1].recpt[1]=-1;

PIC[5].theAUT[0].recpt[0]=1;PIC[5].theAUT[0].recpt[0]=-1;
PIC[5].theAUT[1].recpt[0]=0;PIC[5].theAUT[1].recpt[1]=2;
PIC[5].theAUT[2].recpt[0]=1;PIC[5].theAUT[2].recpt[0]=-1;

Mask[0].T[0]=1;Mask[0].T[1]=1;Mask[0].T[2]=1;
Mask[1].T[0]=1;Mask[1].T[1]=1;
Mask[2].T[1]=1;Mask[2].T[3]=1;Mask[2].T[2]=1;
Mask[3].T[1]=1;Mask[3].T[4]=1;Mask[3].T[2]=1;
Mask[4].T[0]=1;Mask[4].T[1]=1;
Mask[5].T[0]=1;Mask[5].T[1]=1;Mask[5].T[2]=1;

run Cont(0);run Cont(1);run Cont(2);run Cont(3);run Cont(4);
run A;run C;run B;run Chan0;run Chan1;
}
```

7.3.2 A step-by-step execution of the Morse code example

Here is a step-by-step execution of the Morse code example's execution as it is presented in Figure 5.6.

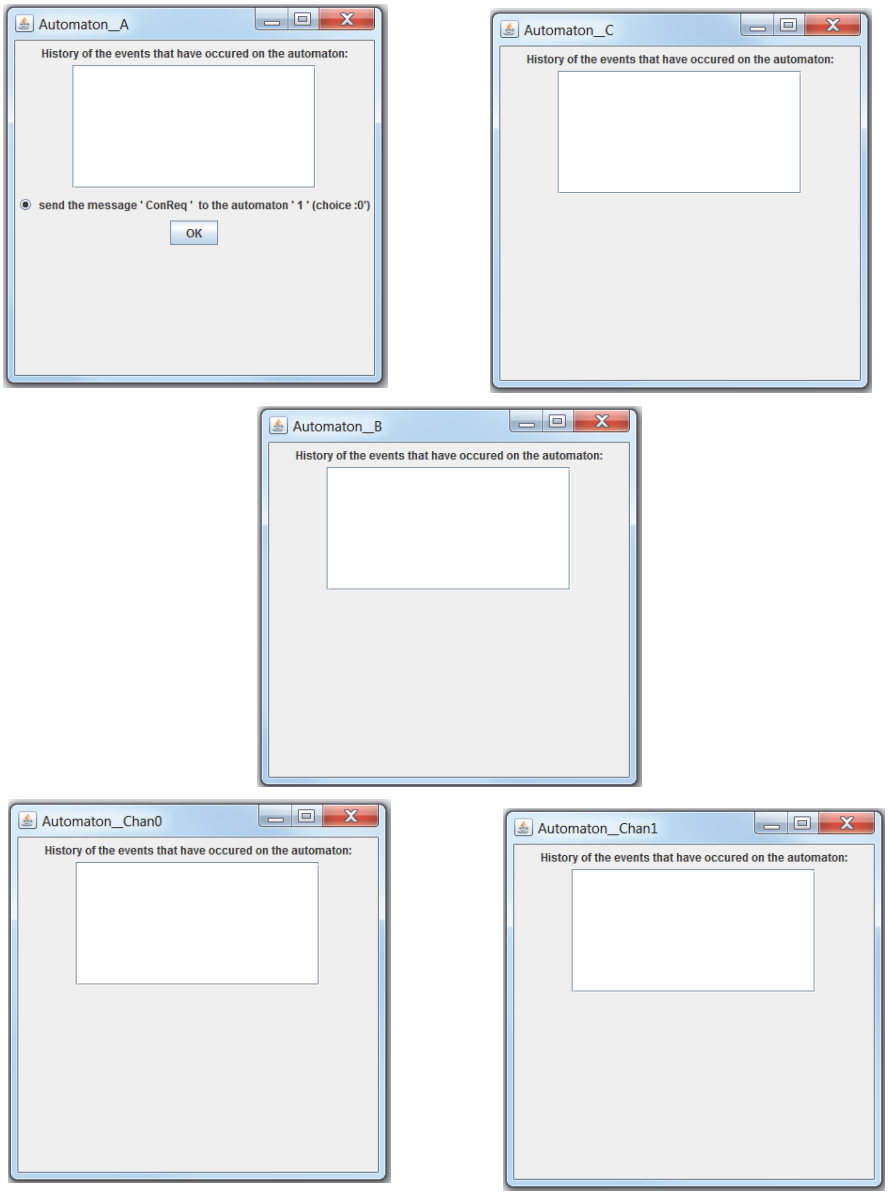


Figure 7.4: The status of the frames presenting the automata of the Morse Code example

We can see that at this level the only possible action can be done by the Automaton A which can send the message “ConReq” to the automaton C.

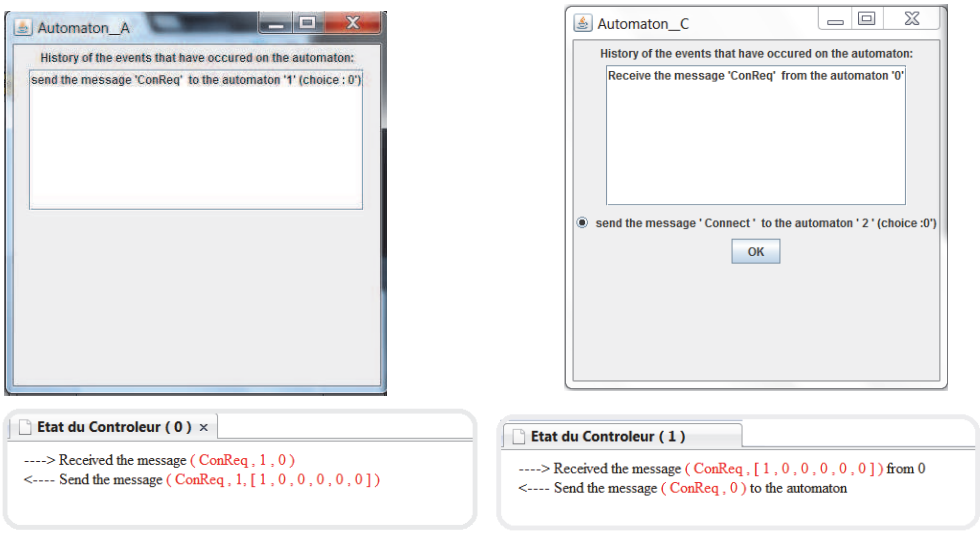


Figure 7.5: A sends ConReq to C

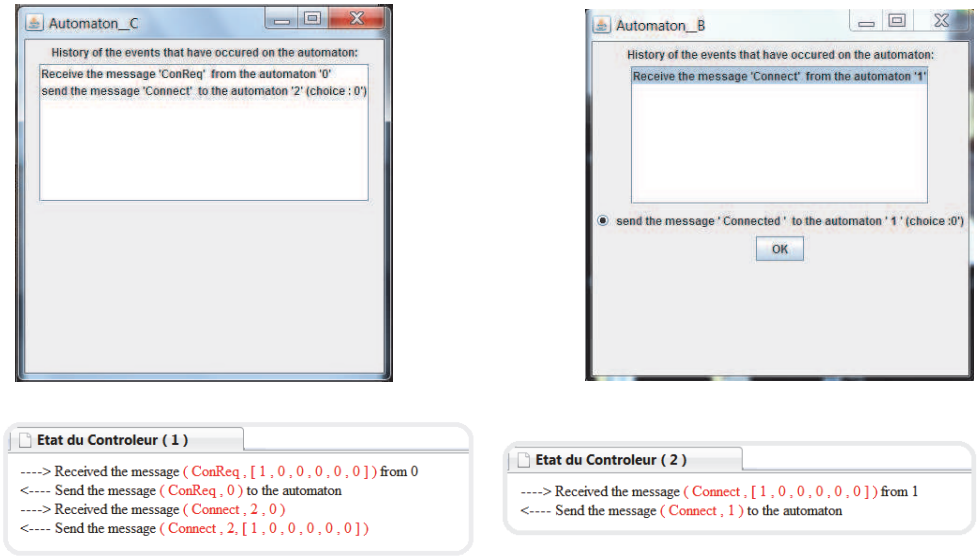


Figure 7.6: C sends Connect to B

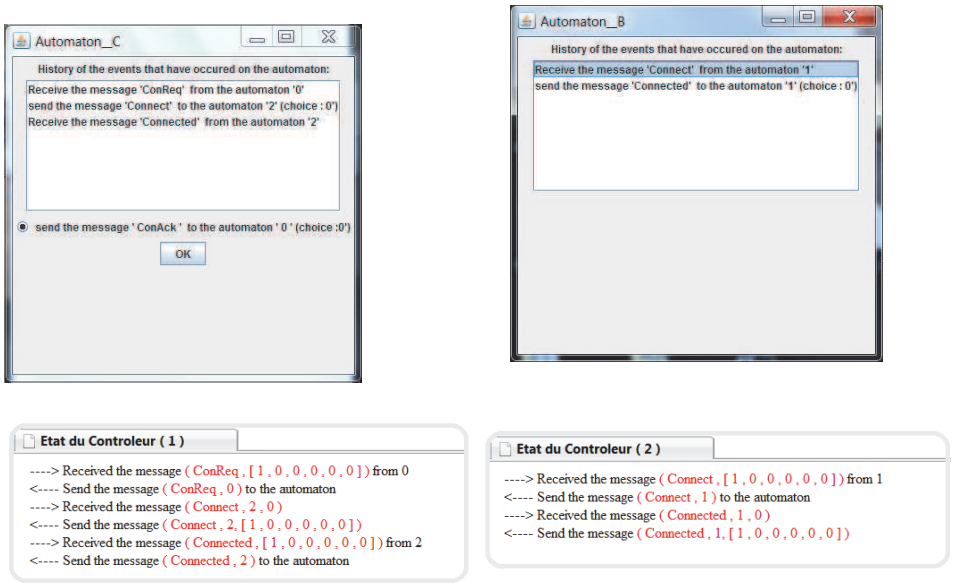


Figure 7.7: B sends Connected to C

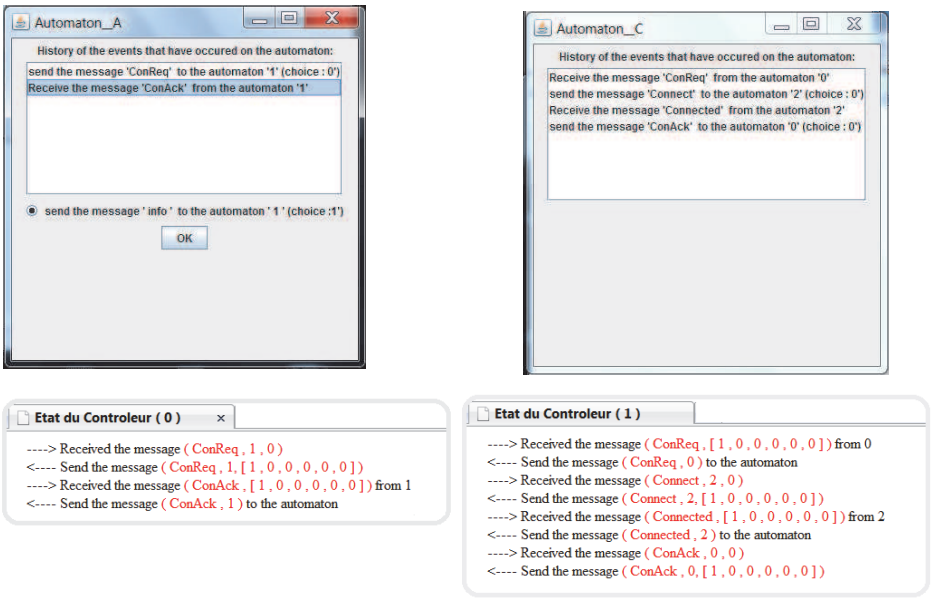


Figure 7.8: C sends ConAck A

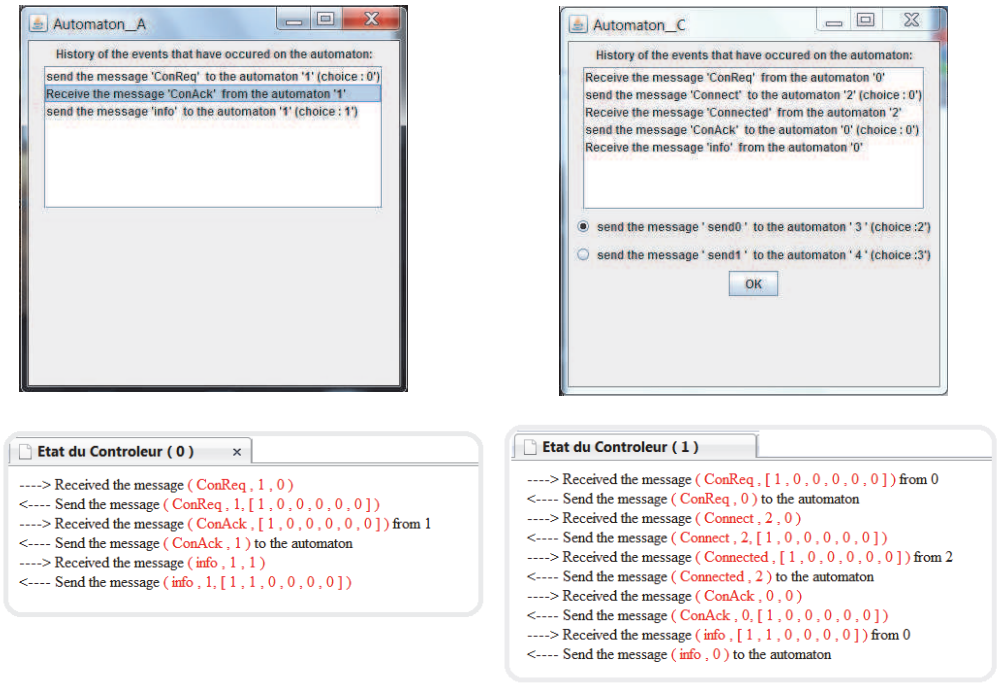


Figure 7.9: A sends Info to C

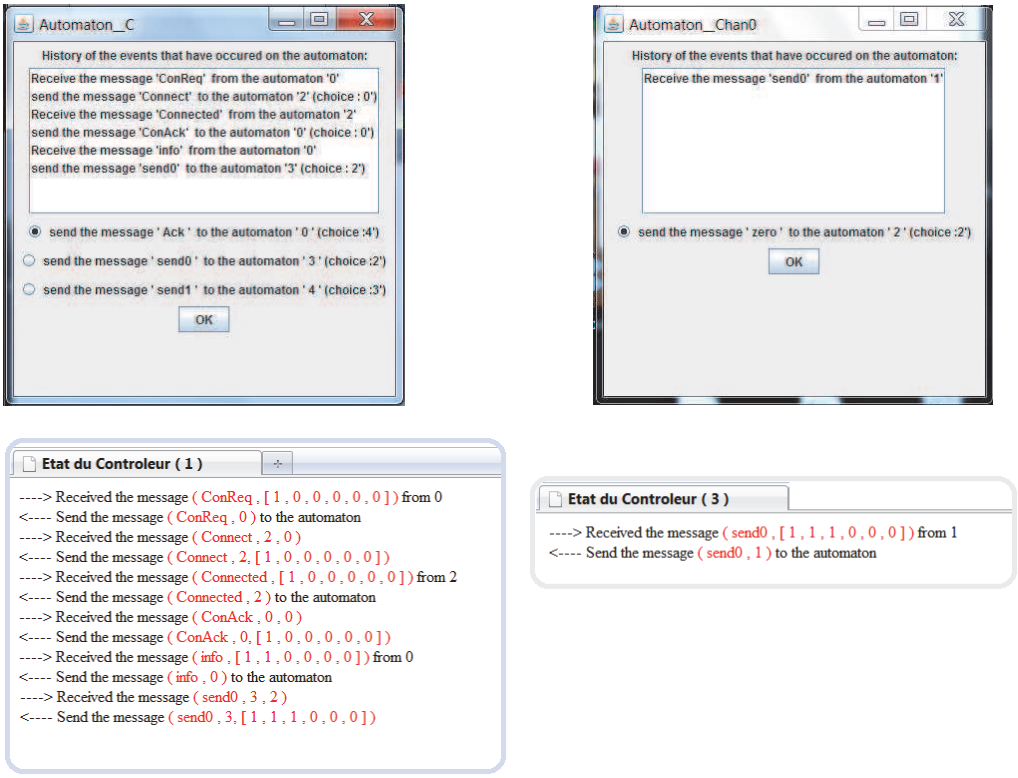


Figure 7.10: C sends send0 to Chan0

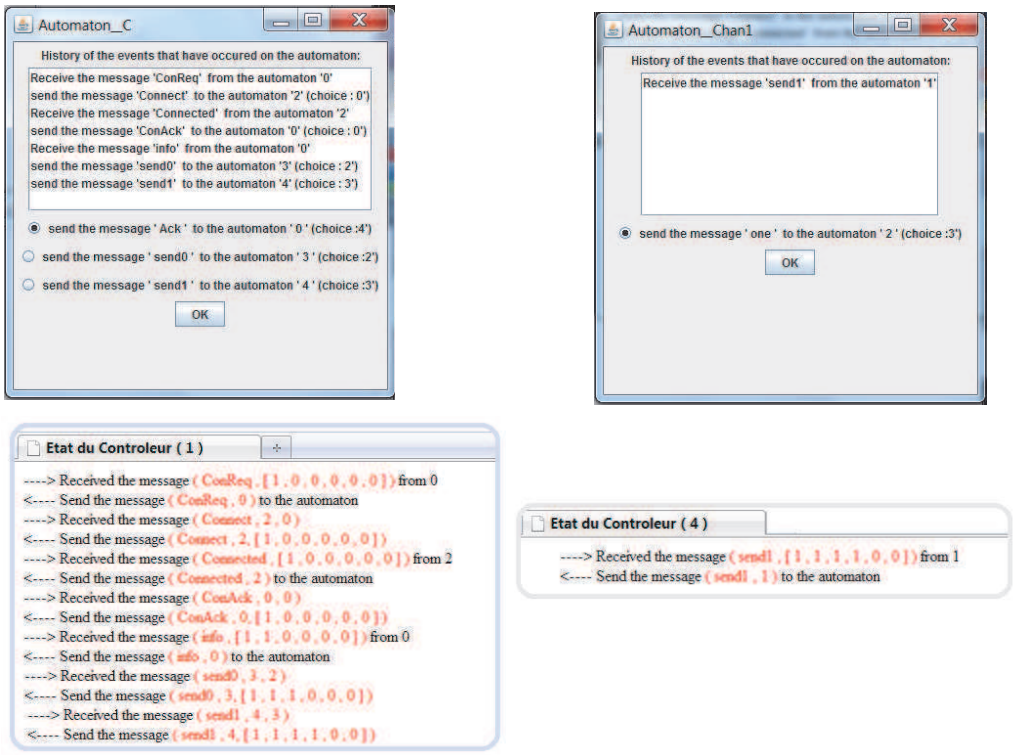


Figure 7.11: C sends send1 to Chan1

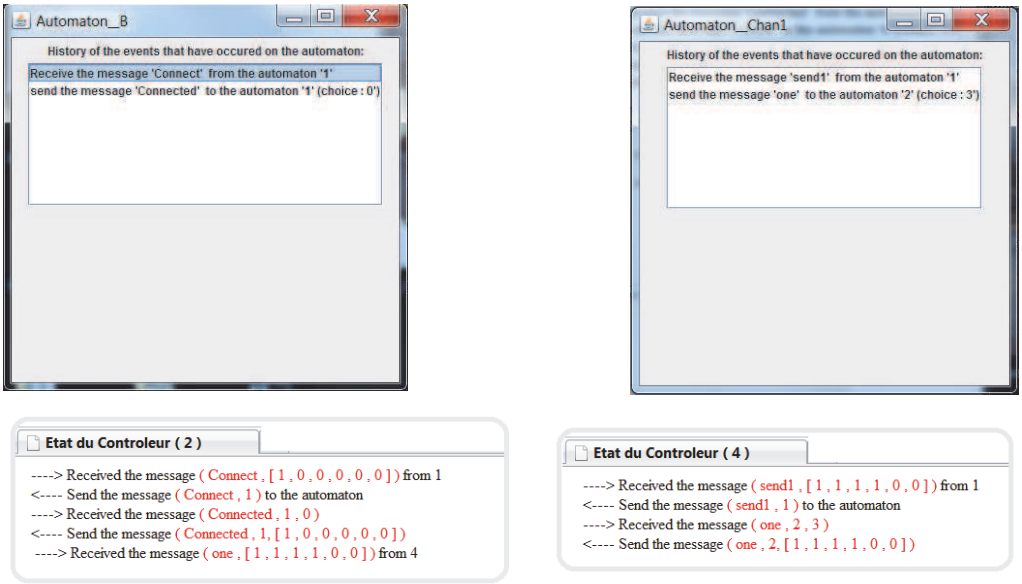


Figure 7.12: Chan1 sends one to B

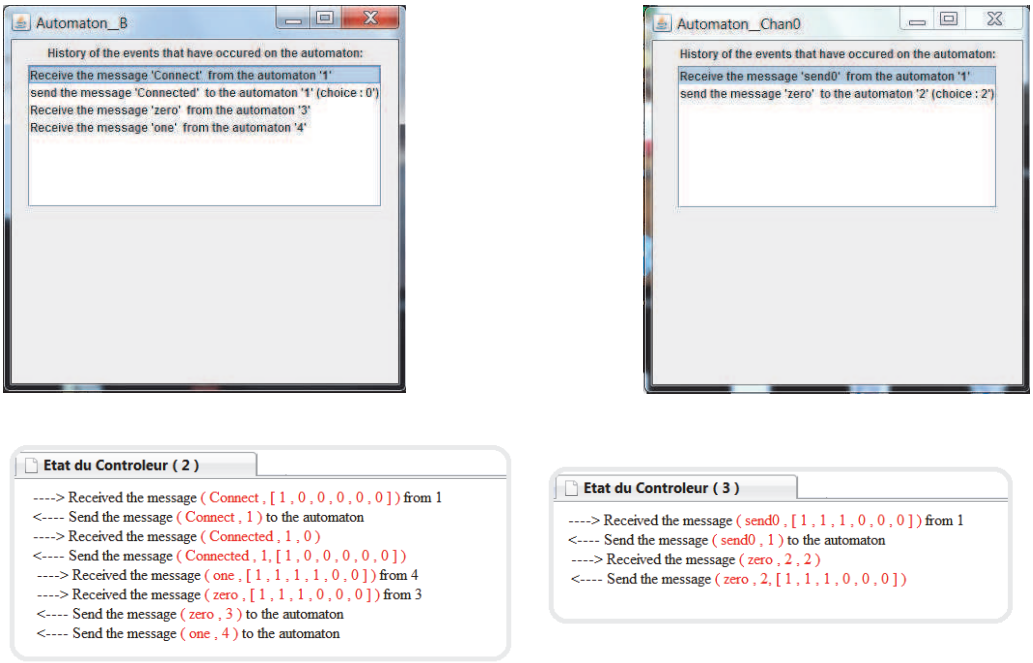


Figure 7.13: Chan0 sends zero to B

7.3.3 The generated Prolog Code for the localisation of the toaster example

The code presented below is the prolog code generated to solve the localisation of the toaster example. Each bMSC M_i present in the HMSC will be concerned by three variables X_i,Y_i, and C_i. X_i is for the leader process that we chose, Y_i is for the set of active processes, and C_i designat the cost of the changes we make for the original bMSC. Having $|I|$ processes in the HMSC, the value of $X_i \in [2^0, 2^{|I|}]$. Y_i is then the sum of the values of the active processes in the bMSC. C_i is the cost of the changes based on the cost function "cost". the cost function is the one described in chapter 4. the predicate "w_in_c(A1,B2, R)" tells if a set of processes B2 contains the process B2. the predicate "w_eq((A1,B1), (A2, B2))" tells if the processes A1 and A2 are the same. these two predicates are used mainly to define the original sets of processes in each bMSC and to define the constraints to respect between the different bMSCs to have a local HMSC. We bound the costs C_i and there bound are processed and generated by SOFAT based on the HMSC's specification. the solving algorithm using labeling and bisect, is described in chapter 4.

```
hmsc((X0,Y0),(X1,Y1),(X2,Y2),(X3,Y3),(X4,Y4),C0,C1,C2,C3,C4,F)  :-
% Encoding singleton with bitwise data structure
list_to_fdset([1,2,4],FD_SET),
X0  in_set FD_SET,
X1  in_set FD_SET,
X2  in_set FD_SET,
X3  in_set FD_SET,
X4  in_set FD_SET,
% Encoding subsets with bitwise data structure
domain([Y0,Y1,Y2,Y3,Y4],0 , 7),
 % *************  CSP Constraints :  *************
 %The domains of variables must contain at least the initial
 %set of active processes
w_in_c(1, Y0 ,1),
w_in_c(2, Y0 ,1),
w_in_c(4, Y0 ,1),

w_in_c(1, Y1 ,1),
w_in_c(2, Y1 ,1),
w_in_c(4, Y1 ,1),

w_in_c(1, Y2 ,1),
w_in_c(2, Y2 ,1),
w_in_c(4, Y2 ,1),

w_in_c(1, Y3 ,1),
w_in_c(2, Y3 ,1),
w_in_c(4, Y3 ,1),

w_in_c(1, Y4 ,1),
w_in_c(2, Y4 ,1),
w_in_c(4, Y4 ,1),


% Each Xi shoul belong to Yi
w_in_c(X0, Y0 ,1),
w_in_c(X1, Y1 ,1),
```

```
w_in_c(X2, Y2 ,1),
w_in_c(X3, Y3 ,1),
w_in_c(X4, Y4 ,1),

%The constraints between variables :
w_in_c(X1 , Y0, 1),
w_eq((X1,Y1) , ( X3,Y3)),
w_in_c(X2 , Y1, 1),
w_in_c(X4 , Y1, 1),
w_eq((X2,Y2) , ( X4,Y4)),
w_in_c(X0 , Y2, 1),
w_in_c(X0 , Y3, 1),
w_in_c(X3 , Y4, 1),
w_in_c(X1 , Y4, 1),

%The constraints of costs : with theta = 0.3
cost(a0,X0,Y0,C0,0.3),
cost(a1,X1,Y1,C1,0.3),
cost(a2,X2,Y2,C2,0.3),
cost(a3,X3,Y3,C3,0.3),
cost(a4,X4,Y4,C4,0.3),
sum([C0,C1,C2,C3,C4],#=, F),
C0 #> -1 ,
C0 #< 3 ,
C1 #> -1 ,
C1 #< 3 ,
C2 #> -1 ,
C2 #< 3 ,
C3 #> -1 ,
C3 #< 3 ,
C4 #> -1 ,
C4 #< 3 ,

labeling([bisect,minimize(F),time_out(60000,Flag)],
[X0,X1,X2,X3,X4,C0,C1,C2,C3,C4]).
w_eq((A1,B1), (A2, B2)) :-
A1 #= A2.
```

```
w_in_c(A1,B2, R) :-
R #<=> ((B2 / A1) mod 2 #= 1).
card(Y,Card):-
Y#=A0*1  +  A1*2  +  A2*4 ,
domain([A0 , A1 , A2  ], 0, 1),
sum([A0 , A1 , A2  ], #= ,Card).
iter([X],[1]):-!.
iter([X|Xs],[M|Ls]):-
iter(Xs,Ls),
Ls=[N|_],
M is 2*N.
cost(A,X,Y,C,Theta):-
domainBase(A,Ybase),
card(Ybase,CardBase),
card(Y,CardNew),
Nb_Processes #= CardNew - CardBase,
numberOfMinimum(A,Nb_minim),
w_in_c(X, Ybase, C_dec1),
the_minimum_local(A,MiniLocal),
w_in_c(X, MiniLocal, C_dec2),
C_dec #= (C_dec1 + (1-C_dec2)) / 2,
C #= Nb_minim + Nb_Processes*(Theta+1) -1 + C_dec.
```

Bibliography

[1] Series z: Languages and general software aspects for telecommunication systems. Editor. 2011.

[2] M. Abdalla, F. Khendek, and G. Butler. New results on deriving SDL specifications from MSCs. In *SDL Forum*, pages 51–66, 1999.

[3] R. Abdallah, C. Jard, and L. Hélouët. Distributed implementation of message sequence charts. *Software and Systems Modeling*, page to appear, 2013.

[4] S. Akshay, M. Mukund, and N.K. Kumar. Checking coverage for infinite collections of timed scenarios. In *CONCUR'07*, pages 181–196, 2007.

[5] A. Albreshne, P. Fuhrer, and J. Pasquier-Rocha. Web services technologies: State of the art. Technical Report no 09-04, Department of Informatics, University of Fribourg, Switzerland, September 2009. http://diuf.unifr.ch/drupal/softeng/sites/diuf.unifr.ch.drupal.softeng/files/file /publications/internal/WP09-04.pdf.

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

[7] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Trans. Softw. Eng.*, 29(7):623–633, July 2003.

[8] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theor. Comput. Sci.*, 331(1):97–114, February 2005.

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

[10] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 114–129, London, UK, UK, 1999. Springer-Verlag.

[11] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.

[12] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context – motorola case study. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 476–491, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] N. Baudru and R. Morin. Safe implementability of regular message sequence chart specifications. In *SNPD*, pages 210–217, 2003.

[14] N. Baudru and R. Morin. Synthesis of safe message-passing systems. In *Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, FSTTCS'07, pages 277–289, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97*, pages 259–274. Springer, 1997.

[16] 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.

[17] D. Brand and P. Zafiropoulo. On communicating finite state machines. Technical Report no RZ1053, IBM Zurich Research Lab, 1981.

[18] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester, 1996.

[20] B. Caillaud, P. Darondeau, L. Hélouët, and G. Lesventes. Hmscs as partial specifications ... with pns as completions. In *MOVEP*, pages 125–152, 2000.

[21] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, pages 191–206, 1997.

[22] M. L. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. In *Proc. 8th International Conf. on Model Driven Engineering Languages and Systems, Oct*, pages 2–7, 2005.

[23] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[24] H. Dan, R. M. Hierons, and S. Counsell. A framework for pathologies of message sequence charts. *Inf. Softw. Technol.*, 54(11):1283–1295, November 2012.

[25] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[26] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[27] Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[28] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.

[29] B. Genest. Compositional message sequence charts (cmscs) are better to implement than mscs. In *TACAS*, pages 429–444, 2005.

[30] B. Genest, D. Kuske, and A. Muscholl. A kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.*, 204(6):920–956, June 2006.

[31] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, June 2006.

[32] J. Grabowski. *Test Case Generation and Test Case Specification Based on Message Sequence Charts*. PhD thesis, Dissertation, Universität Bern, Institut für Informatik, Februar 1994, February 1994.

[33] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[34] O. Haugen. Comparing uml 2.0 interactions and msc-2000. In *Proceedings of the 4th international SDL and MSC conference on System Analysis and Modeling*, SAM'04, pages 65–79, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] O. Haugen. Message sequence charts (msc). In *ITU, Z.120*, Editor. 1999, ITU-T: Geneva. p. 126.

[36] L. Hélouët, R. Abdallah, and D. Bhatia. SOFAT : Scenario formal analysis toolbox. Technical report, 2011. `www.irisa.fr/distribcom/Prototypes/SOFAT/`.

[37] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from hmscs. In *5th International Workshop on Formal Methods for Industrial Cr itical Systems (FMICS)*, Berlin, April 2000. GMD FOKUS.

[38] L. Hélouët, C. Jard, and B. Caillaud. An effective equivalence for sets of scenarios represented by hmscs. Technical Report 3499, INRIA, September 1998.

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

[40] G. J. Holzmann, D. A. Peled, and M. H. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 2:86–95, 1997.

[41] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[42] IBM. Telelogic tau, version 4.2. Technical report, http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/index.jsp?topic=/com.ibm.help.download.tau.doc/topics/taudownload42.html.

[43] ITU-T. Z.100 : Specification and description language (SDL). Technical report, International Telecommunication Union, 2011.

[44] R. J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.).* Wiley, 2008.

[45] I. Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach.* Addison–Wesley, Wokingham, England, 1995.

[46] C. Jervis. Message sequence charts (msc). In *ITU, Z.120*, Editor. 2004, ITU-T: Geneva.

[47] O. Kluge. Petri nets as a semantic model for message sequence chart specifications, 2004.

[48] K. Koskimies and E. Mäkinen. Automatic synthesis of state machines from trace diagrams. *Softw. Pract. Exper.*, 24(7):643–658, July 1994.

[49] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, DIPES '98, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[50] R. L. Krikhaar and J. G. Wijnstra. Product development with the building block method , a process perspective. Technical report, Philips Research Information and Software Technology, 1994-12-06.

[51] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[52] S. Leue and P. B. Ladkin. Implementing and verifying msc specifications using promela/xspin. In *Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*, volume 32 of *DIMACS*, pages 65–89, 1997.

[53] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing room models from message sequence chart specifications, 1998.

[54] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing software architecture descriptions from message sequence chart specifications. In *ASE*, pages 192–195, 1998.

[55] H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In Jon Whittle, Leif Geiger, and Michael Meisinger, editors, *SCESM*, pages 5–12. ACM, 2006.

[56] M. Lohrey. Safe realizability of high-level message sequence charts. In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 177–192, London, UK, UK, 2002. Springer-Verlag.

[57] M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.*, 309(1):529–554, December 2003.

[58] B. Lüdemann. *Synthesis of Human-readable Statecharts from Sequence Diagrams in the ROOM Environment.* 2005.

[59] N. Mansurov and D. Zhukov. Automatic synthesis of sdl models in use case methodology. In *SDL Forum*, pages 225–240, 1999.

[60] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[61] 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.

[62] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[63] S. Mauw and M. A. Reniers. Refinement in interworkings. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 671–686, London, UK, UK, 1996. Springer-Verlag.

[64] Sun Microsystems. Rpc: Remote procedure call protocol specification. RFC 1050, Apr 1988.

[65] B. Mitchell, R. Thomson, and C. Jervis. Phase automaton for requirements scenarios. In Daniel Amyot and Luigi Logrippo, editors, *FIW*, pages 77–84. IOS Press, 2003.

[66] R. Morin. Recognizable sets of message sequence charts. In *STACS 2002, LNCS 2030*, pages 523–534. Springer, 2002.

[67] M. Mukund, K. N. Kumar, and M. A. Sohoni. Synthesizing distributed finite-state systems from mscs. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR '00, pages 521–535, London, UK, UK, 2000. Springer-Verlag.

[68] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.

[69] A. Muscholl and D. Peled. Deciding properties of message sequence charts. In *Scenarios: Models, Transformations and Tools*, pages 43–65, 2003.

[70] A. Muscholl, D. Peled, and Z. Su. Deciding properties for Message Sequence Charts. In *FoSSaCS*, volume 1378 of *LNCS*, pages 226–242, 1998.

[71] OMG. *UML 2.0 : Unified Modeling Language*. Object Management Group, August 2005.

[72] J. Padberg, O. Kluge, and H. Ehrig. Modeling train control systems: From message sequence charts to petri nets, 2000.

[73] C.A. Petri. *Kommunication mit Automaten*. PhD thesis, Technischen Hoschule Darmstadt, 1962.

[74] 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.

[75] M. A. Reniers. Message sequence chart: Syntax and semantics. Technical report, Faculty of Mathematics and Computing, 1998.

[76] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[77] E. Rudolph. Message sequence charts (msc). In *ITU, Z.120*, Editor. 1996, ITU-T: Geneva. p. 78.

[78] E. Rudolph, J. Grabowski, and P. Graubmann. Towards a harmonization of uml-sequence diagrams and msc. In *SDL Forum*, pages 193–208, 1999.

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

[80] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[81] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *ICSE*, pages 188–197, 2001.

[82] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in Message Sequence Chart specifications. In *ESEC / SIGSOFT FSE*, pages 74–82, 2001.

[83] 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, January 2004.

[84] H. Vafaie. Application of petri-nets in the hermes data flow machine: an overview. In *Circuits and Systems, Proceedings of the 32nd Midwest Symposium*, 1989.

[85] P. Van Hentenryck, V.A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3):139–164, 1998.

[86] VERILOG. Objectgeode 4.0, 1997. http://www.control.aau.dk/ henrik/undervisning/embedd/tutorial.pdf.

[87] G. Von Bochmann. *Finite State Description of Communication Protocols.* Reports Montreal Univ Canada. 1976.

List of Publications

International Journal

[1] R. Abdallah, C. Jard, and L. Hélouët, Distributed implementation of message sequence charts. *Software and Systems Modeling*, page to appear, 2013.

[**2**] B. Daya, H. Akoum, and R. Abdallah, Vehicule Detection Using Horizon Base Approaches for the Real Time System , *International Journal of Sciences and Techniques of Automatic control & computer engineering IJ-STA*, Volume 4, No 2, pp. 1284-1297, December 2010.

International Conference

[1] R. Abdallah, A. Gotlieb, L. Hélouët, and C. Jard, Scenario realizability with constraint optimization, *FASE 2013*, pp. 194-209, March 2013.

[2] R. Abdallah and C. Jard, An experiment in automatic generation of protocols from HMSCs, *Notere 2011*, pp. 1-8, May 2011.

[**3**] M. Dib, R. Abdallah, A. Caminada, Arc-consistency in Constraint Satisfaction Problems: A survey, *2nd International Conference on Computational Intelligence, Modeling and Simulation*, Bali, Indonesia, 2010.

[**4**] B. Daya, H. Akoum, R. Abdallah, and L. Prevost. Vehicle Detection Using New Approach for the Real Time System. *International Conference on Sciences and Techniques of Automatic control & computer engineering*, 2009.

National Conference

[**1**] M. Dib, A. Caminada, and R. Abdallah, Tabu-Ng: Une Approche De Résolution Hybride Pour La Coloration De Graphes".*ROADEF 11*, Saint-Etienne, Mars 2011.