A simulation model for Message Sequence Charts*

Loïc Hélouët†
IRISA, Campus de Beaulieu
35042 Rennes Cedex
helouet@irisa.fr

This article describes a simulation framework for High-level Message Sequence Charts based on a graph grammar representation of MSCs. Normalized grammars allow for asynchronous or synchronized executions of HMSCs, and helps detecting process divergence.

1. Introduction

Specification of distributed systems is a difficult task. Unspecified behaviours can be introduced by unexpected concurrency, and communication media play an important role in the correctness of a protocol (distributed program won't have the same behaviour with asynchronous or synchronous communication media). Message Sequence Charts (MSCs) allow for the specification of distributed systems. They are clear, easy to use, and abstract the communication media from the conception task. MSCs describe a causal order between events of a scenario, and are defined by two types of graphs. Basic Message Sequence Charts (called bMSCs hereafter) define single scenarios. The composition of bMSCs is then done by High-Level Message Sequence Charts, a kind of high-level graph allowing for sequencing and exclusive choice between MSCs. HMSCs are far less intuitive, and can be ambiguous, more especially when a choice between two scenarios is distributed. The official semantics of choices in HMSCs considers that the first instance reaching a choice is responsible for the decision of the scenario to execute, and that all other instances have to perform the same choice. This can result in the storage of an infinite set of history variables (as shown in [1]). A distributed choice can also be treated as a consensus between processes, which involves that the first process able to choose a scenario has to wait for the other's answers. With that interpretation, choices act as synchronizations.

According to the assumed meaning of a choice, the execution of a system specified by an HMSC can be very different. HMSCs appear to be less intuitive than expected. This reinforces the need for formal manipulations of MSCs. Techniques such as model-checking can be used for ensuring the system does not violate critical properties. Unfortunately, MSCs describe infinite behaviours, with infinite state-space, which makes most of the model-checking techniques useless. Another technique for increasing the confidence in the system is simulation. By simulating the behaviour of a distributed program, one can quickly detect undesirable behaviours at an early stage of the specification.

*This work was partially supported by Alcatel within the REUTEL project
†INRIA

This article proposes a simulation framework for Message Sequence Charts based on the graph grammar representation of MSCs described in [4]. The normalized graph grammars turn out to be a good model for simulation, by reducing the time needed for searching the enabled actions of a system in a given state. Furthermore, the growth of the state size is a good indicator of a divergent specification.

This article is organized as follows: sections 2 defines a partial order semantics for HMSCs. Section 3 describes a graph grammar representation of MSCs and its properties. Section 4 gives an operational semantics for executing a normalized grammar. Section 5 is a complete example. Section 6 concludes, and gives perspectives for further work.

2. Partial order representation of MSCs

This section describes a partial order semantics for MSCs, very close to [6,5]. Partial order semantics models independence of events, while semantics such as [7] models concurrency through an interleaving of actions.

2.1. bMSCs as partial orders

A bMSC (standing for basic MSC) defines a simple scenario, ie an abstraction of a system behaviour. Within bMSCs, processes are called *instances*, and are represented by a vertical axis, along which events are put in a top down order. Message exchanges are represented by arrows labeled by message names from the emitting to the receiving instance. No assumption whatsoever is made about the communication medium. The MSC events can be: communication event (sending, receiving a message), timer events (setting, resetting, or timeout), instance events (creation, stop) or internal actions.

A bMSC defines a precedence relation between events: the emission of a message precedes its reception, and for any event e, all events situated upper than e on the same instance axis are predecessors of e. The order on the axis can be relaxed in some parts of the instance called *co-regions*. These co-regions are represented by dashed parts of the instance axis. Events situated in a co-region are not necessarily concurrent: their order is not specified yet, or is not important for the specification.

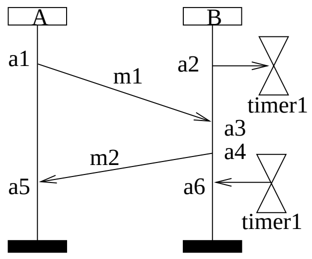

Figure 1. An example bMSC.

Let us consider a bMSC M describing the behaviour of a finite set of instances, called I hereafter. Two events performed by the same instance are ordered according to their coordinates on the instance axis, provided they are not in a co-region. If they are performed by different instances, they are ordered if and only if they are separated by at least one message exchange. So, M defines a partial order between events, and can be formalized by a poset $M =< E, \leq, \alpha, A, I >$ where E is a set of events, $\leq$ is a partial order relation (reflexive, transitive and antisymmetric binary relation) on E called *causal dependence relation*, A is a set of atomic actions names, and α is a labeling function from E to $A \times I$, associating an action name and an instance to each event. In Figure 1, a_1 and a_4 are ordered, but a_1 and a_2 are not.

2.2. HMSCs as partial order families

bMSCs only define simple finite scenarios. Designing communicating systems requires modelling more complex structures. High-level Message Sequence Charts (HMSCs) is a higher-level notation that allows for the composition of bMSCs. The usual syntax for HMSC, called MSC'96 [9] includes sequential and parallel composition of MSCs, loops, choices and inline expressions. This paper only deals with a subset of MSC'96, without inline expressions (which can be formalized by means of higher level descriptions) or parallel composition. Consequently, an HMSC can be seen as a directed graph, the nodes of which are start nodes, end nodes, bMSCs, connection symbols, conditions, or references to other HMSCs. An exemple is shown in Figure 8.

The official semantics of **sequential composition** of bMSCs is defined by a partial order concatenation operation. Let $M_1 =< E_1, \leq_1, \alpha_1, A_1, I_1 >$ and $M_2 =< E_2, \leq_2, \alpha_2, A_2, I_2 >$ be two MSCs. The *concatenation* of M_1 and M_2, written $M_1 \circ M_2$ is defined by:

$$M_1 \circ M_2 =< E_1 \cup E_2, \leq_{M_1 \circ M_2}, \alpha_1 \cup \alpha_2, A_1 \cup A_2, I_1 \cup I_2 >, \text{ where:}$$

$$\begin{aligned} \leq_{M_1 \circ M_2} \;=\; & \leq_1 \cup \leq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 \mid \\ & \exists (e'_1, e'_2) \in E_1 \times E_2 \wedge \phi_1(e'_1) = \phi_2(e'_2) \wedge e_1 \leq_1 e'_1 \wedge e'_2 \leq_2 e_2\} \end{aligned}$$

More intuitively, a concatenation "glues" together two MSCs along their common instance axis. Note that this "weak sequential composition" imposes no synchronization barrier between events of M_1 and events of M_2. This means that in $M_1 \circ M_2$, all events of M_2 may be finished before any event of M_1 is performed. Another vision of sequential composition is "strong sequential composition", which imposes that all event of M_1 must be executed before executing events of M_2. This strong sequential composition is often met in the MSC-related literature, and reduces HMSCs to finite state machines. Yet, the official semantics proposed in norm Z.120 [10] is weak sequential composition.

The **alternative** between two MSCs is an exclusive choice between scenarios. HMSC P_0 in Figure 2-a defines two possible scenarios, represented in Figure 2-b.

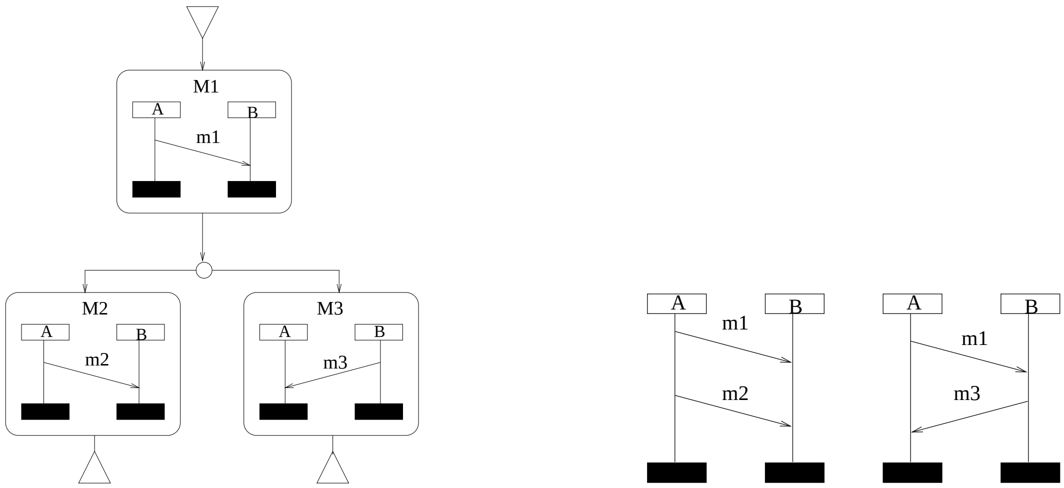

Figure 2. a)HMSC P_0 (with non-local choice). b)Scenarios defined by P_0.

A *path* of an HMSC is a sequence of nodes $n_1.n_2...n_k$ such that $\forall i \in 1..k, n_{i+1}$ is a successor node of n_i. The set of scenarios defined by an HMSC is the sequential composition of partial orders met along the paths of the HMSC. As HMSCs may contain cycles, and as alternatives allows for multiple branchings, an HMSC may describe an infinite number of different path, and infinite paths. So, the set of scenarios defined by an HMSC is a possibly infinite set of possibly infinite partial orders. This set will be called partial order family (or POF for short) hereafter. Let H be an HMSC, the POF defined by H is:

$POF(H) = \{M_1 \circ M_2 \circ ...M_n | \exists \text{ a path in } H \text{ going through } M_1, M_2...M_n\}$

Definition 1:

An *execution* $w = e_0e_1...$ of an HMSC is a possibly infinite word of E^*. w is *consistent* with respect to an order $M =< E, \leq, \alpha, A, I >$ if and only if:

- $e_0 \in min(M)$ the first event executed is in the minimal events of M,
- $\forall i, j | i < j, e_j \not\leq e_i$ (there is no contradiction between the total order defined by w and the order $\leq$).

Let us consider HMSC P_0 in Figure 2-a. P_0 describes two possible scenarios: $M_1 \circ M_2$ and $M1 \circ M3$. To be consistent with the specification, an execution of P_0 will have to be consistent with the order defined by $M_1 \circ M_2$ or by $M_1 \circ M_3$. Let us note $!m$ the sending of a message m, and $?m$ for it corresponding reception. Consistent executions of P_0 are $w_1 =!m_1.?m_1.!m_2.?m_2$, $w_2 =!m_1.!m_2.?m_1?m_2$ and $w_3 =!m_1.?m_1.!m_3.?m_3$. After the execution of $!m1.?m1$, instance A can decide to perform scenario M_2, and instance B can decide to perform scenario M_3. This situation is be called a *non-local choice*[1]. As the decision to perform M_2 or M_3 is distributed, this can not be done without a synchronization between A and B. Non-local choices can be implemented by means of hidden synchronizations, or distributed consensus mechanisms. However, the simulation of a scenario with non-local choice will suppose that a solution has been implemented, and will only consider conflicts at an abstract level.

HMSCs describe infinite families of potentially infinite partial orders, but these orders may have common prefixes, and are based on finite sets of patterns. We want to find a finite representation of POFs that would preserve the partial order representation, make conflicts apparent, and allow for formal manipulations. Next section gives an event structure semantics to HMSCs equivalent to the POF semantics, and shows that these event structures can be represented finitely by means of graph grammars.

3. Graph grammar representation of HMSCs

This section describes a finite representation of POFs by graph grammars, the unfolding of which are representations of POFs by means of event structures. First, event structures are introduced as a "compact" way of describing POFs, then a short introduction to graph grammars is given and the calculus of a grammar from an HMSC is briefly described. This section recalls results of [4], so proofs are omitted.

3.1. Event Structures

A prime event structure (ES for short) is a 6-tuple $< E, \leq, \sharp, \alpha, A, I >$ where E, A , I, $\leq$ and α have the same meaning than in previous sections, and $\sharp$ is a symmetric anti-reflexive binary relation called *conflict relation*, defining the pair of events that can not appear in the same execution of an ES, and such that:

$\forall e \in E, \forall e' \in E, e \sharp e' \Leftrightarrow \forall e'' \in E, (e' \leq e'' \Rightarrow e \sharp e'')$ (the conflicts are inherited through causality relations).

An event structure defines a domain of *configurations*, that can be seen as possible states of the system. A configuration is a subset C of E that is conflict-free ($\forall e \in C, \nexists e' \in C \mid e \sharp e'$), and downward-closed ($\forall e \in C, e' \leq e \Rightarrow e' \in C$). An ES defines a family of orders, which are projections of the order relation on its maximal configurations. $POF(ES) = \{max(< E_i, \leq >) | \forall e, e' \in E_i, e \not\sharp e'\}$. For further reading on event structures, consult [8].

The construction of an event structure form an HMSC is straightforward. The partial order relation is built using the weak sequencing operator on the paths of the HMSC. Conflicts are introduced by choices: two events situated in different suffixes of different paths of the HMSC are conflicting events. So, **an execution of an HMSC must not contain events conflicting in its ES representation**.

An event structure can be represented by a graph which associates a vertex to each event, and a typed edge to each conflict and each pair in the order relation. Infinite behaviours lead to infinite graphs. Unfortunately, the resulting graph is not necessarily a regular graph, which means that it can not always be generated by a graph grammar. A first intuitive solution is to represent only minimal conflicts, and the covering of the causality order. The other conflict and causality edges can be deducted from the graph representation, using the conflict inheritance property and the transitivity of the order relation.

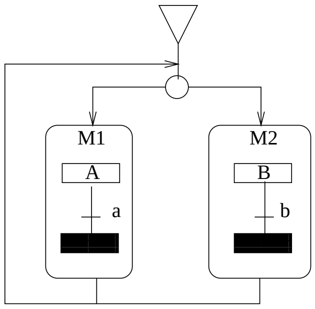

Figure 3. An irregular HMSC.

Even though, graphs obtained from ES may still not be regular: the graph representation for the event structure of HMSC in Figure 3, is the irregular graph in Figure 4-a.

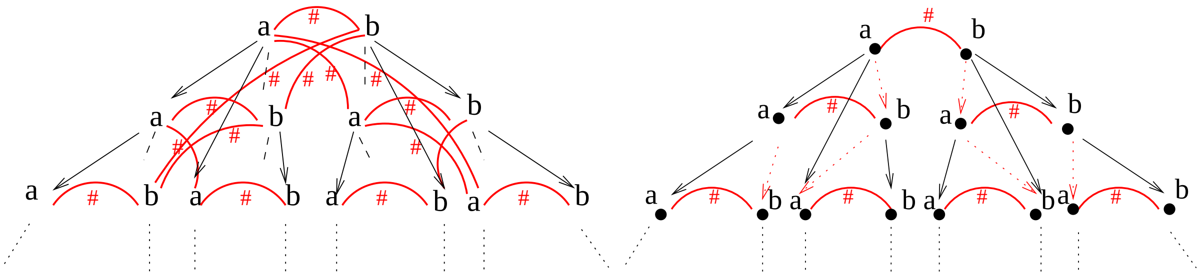

Figure 4. a)Irregular SE graph. b)Regular covering graph.

In order to solve the problem of regularity caused by conflict edges in the graph representation of an event structure, a new type of edge is defined, and is called conflict inheritance edge. This relation will allow an event e' to inherit all conflicts from an event e, without being causally dependent on e.

An ES can be represented by a *covering graph*: $< E, \longrightarrow, \rightsquigarrow, \sharp_c, \alpha, A, I >$, such that:

- $\longrightarrow = \{(e, e') \in \leq \mid e \neq e' \wedge \nexists e'' \in E - \{e, e'\}, e \leq e'' \leq e'\}$ is the covering of $\leq$,
- $\rightsquigarrow$ is an conflict inheritance relation $(\forall (e_1, e_2) \in \rightsquigarrow, \forall e' \in E, e' \sharp e_1 \Rightarrow e' \sharp e_2)$,
- $\sharp_c$ is the set of minimal conflicts, i.e. $\{(e, e') \in \sharp \mid \nexists e'' \wedge (e'' \leq e' \vee e'' \rightsquigarrow e') \wedge e \sharp e''\}$.

The conflict inheritance relation allows for generating regular covering graphs, that can be represented finitely by a graph grammar.

A covering graph of the event structure of Figure 4-a is represented in Figure 4-b. $\longrightarrow$ is represented by simple arrows, $\rightsquigarrow$ is represented by dotted arrows, and $\sharp$ is represented by an edge labeled by $\sharp$.

For the sake of clarity, the labeling part of covering graphs will be omitted, and a covering graph will be noted $G =< E, \longrightarrow, \rightsquigarrow, \sharp >$.

3.2. Graph grammars

Only a short introduction to graph grammars is given. For further reading, [3] may be consulted. A graph grammar is a grammar in which terminal are vertices and edges, and non-terminals are hyperarcs.

An *hypergraph* G is a pair $(T(G), H(G))$, where $T(G)$ is a finite graph, and $H(G)$ is a set of hyperarcs. An *hyperarc* is a word $ls_1..s_n$, which label l belongs to an alphabet L, and where $\{s_i\}$ are vertices of $T(G)$. On the hypergraph represented in Figure 6, the vertices s_4, s_5, s_6 are linked by an hyperarc labeled by A. Such an hyperarc will then be noted $As_5s_6s_4$.

A *graph grammar* consists in an hypergraph G_0 called the *axiom* of the grammar, and of a set $\mathcal{R}$ of rewriting rules. A rewriting rule is a pair (X, H), where X is an hyperarc, and H is the hypergraph rewritten from X's vertices. We will often write this rule $X \triangleright H$. X will be called the *left part* of the rule, and H the *right part*. For an hypergraph G, we note $G \underset{\mathcal{G},X}{\longrightarrow} G'$ if G can be rewritten into G' by replacing X by H in G. A *direct derivation* of an hypergraph G by a rule $(X, R) \in \mathcal{R}$ is an hypergraph G' such that $G' = G_{[X:=R]}$.

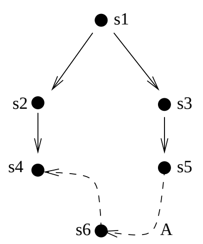

Figure 6: Hypergraph.

We write $G \underset{\mathcal{G},(X,R)}{\longrightarrow} G'$. A sequence of direct derivations $G \underset{\mathcal{G},(X_1,R_1)}{\longrightarrow} G_1 \underset{\mathcal{G},(X_2,R_2)}{\longrightarrow} \ldots \underset{\mathcal{G},(X_n,R_n)}{\longrightarrow} G_n$ is called a *derivation of length n*. We will say that an hypergraph G' is *accessible* from G if there is a derivation leading from G to G'.

3.3. Graph grammar calculus from an HMSC

Section 3.1 has defined a covering graph representation for the POF semantics of an HMSC. As covering graphs are regular, they can be represented by means of graph grammars. These graph grammars can be calculated directly from an HMSC, without developing its POF or it ES representation. As this calculus can be found in [4](p 13-20), we just give a short description. A graph grammar calculus from an HMSC consists in generating rules that will glue together all parts of the covering graph for the SE representation of an HMSC. The right parts of the rules will be either a covering graph without conflict in the case of a single sequencing, and a covering graph with conflicts between minimal events of each bMSC in the case of a choice. The hyperarcs will be sets of maximal events for the order relation. The end nodes will lead to rules that delete hyperarcs. Figure 6 illustrates how MSC features can be transformed into graph grammar rules. From this first step, a normalized grammar can be constructed.

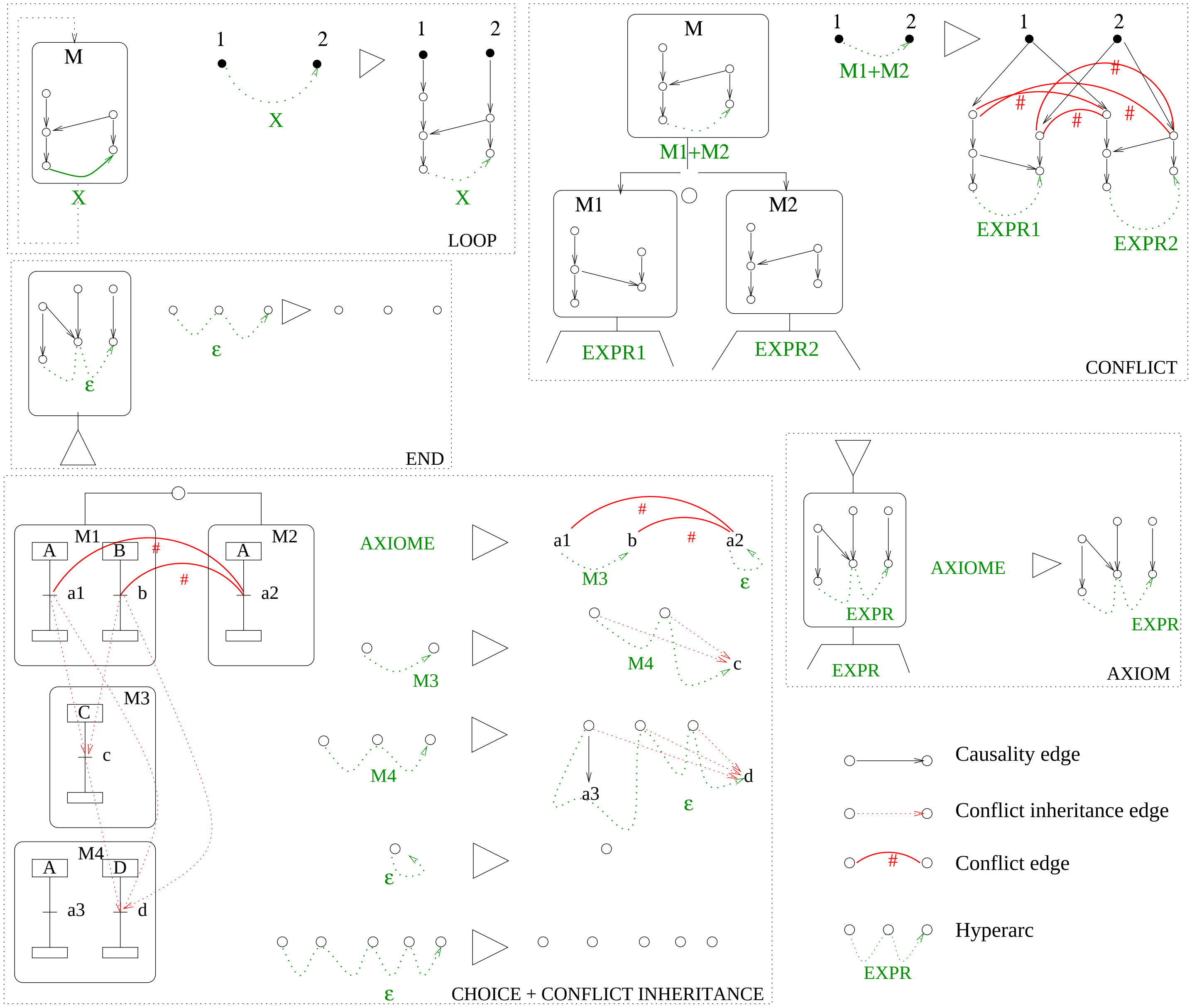

Figure 6. Graph grammar construction from an HMSC.

3.4. Normalized grammars

Once a graph grammar representation of an HMSC has been found, it can be transformed into an equivalent normalized grammar. Right parts of normalized rules contain vertices situated at the same distance. In this article, the distance measure is the number of edges (including causality, inheritance, and conflict edges) needed for reaching a vertex from a set of initial vertices, without consideration for the orientation of the edges. The set of starting vertices is the first event performed on each instance of the HMSC, or if these events are ordered, the minimal events.

Let $\mathcal{G} = (G_0, \mathcal{R})$ graph grammar. $\mathcal{G}$ is normalized if and only if:

- $\mathcal{G}$ is *proper*: $\forall (X, H) \in \mathcal{G}$, any vertex of X is a terminal vertex (it can't be removed by a rule).
- $\mathcal{G}$ is *normal*: $\forall (X, H) \in \mathcal{G}, X$ and H have no common vertices
- $\mathcal{G}$ is *separated*: $\forall (X, H) \in \mathcal{G}$, two hyperarcs of H have no common vertices.
- $\mathcal{G}$ is *uniform*: $\forall (X, H) \in \mathcal{G}$, every vertex of any hyperarc of H is connected to a vertex of X by an edge.

A normal form calculus for graph grammars representing finite branching graphs was given in [2]. This result was adapted for the graph grammar representation of HMSCs in [4]. In order to work on finite branching covering graphs, some causality edges generated inside loops have to be removed from the initial grammar. These edges can still be deduced from inheritance edges, and reintroduced in the covering graph when an hyperarc is unfolded. As these edges can be treated easily during unfolding, they won't be mentioned during the rest of the article.

Furthermore, the graph grammars calculated from HMSCs are *deterministic*, ie for a given hyperarc, there is only one rewriting rule.

Theorem 1:

For a given normalized grammar $\mathcal{G} = (G_0, \mathcal{R})$, and for a given hypergraph $G = (T(G), H(G))$ accessible from the axiom G_0, all predecessors and successors of an event $e \in T(G)$ are written after at most one rewriting step from an hyperarc containing e.

Proof: As a normalized graph grammar is separated, an event e belongs to at most one hyperarc. If e is a vertex of an hyperarc $H \in H(G)$, then as $\mathcal{G}$ is normal, for any hyperarc H' appearing after a rewriting of H, $e \notin H'$. So any edge starting from event e is written during this rewriting step of $\mathcal{G}$. If e isn't a vertex of any hyperarc, then no edge will be connected to e by further rewritings. $\square$

4. Simulation of HMSCs

A simulation of an HMSC H is based on the normalized graph grammar of H. We can distinguish 3 different execution modes, that will be described more precisely later: they are called asynchronous, conflict dependent, and consensus execution modes. The main idea of the simulation is to unfold a limited part of the covering graph containing the next executable events. This graph has to be recalculated after the execution of an event: due to the conflict relations, some events are not executable anymore, and have to be removed. This section defines global system states, and then gives 3 semantics of transitions.

4.1. States

Definition 2:

The set of minimal events of a graph $G =< E, \longrightarrow, \rightsquigarrow, \sharp >$ with respect to a relation $\dashrightarrow \subseteq E \times E$ is noted $min(G, \dashrightarrow) = \{e \in E | \nexists e' \wedge e' \dashrightarrow e\}$.

A *global state* of the system is represented by an hypergraph G, accessible from the axiom of the normalized grammar. This hypergraph represents a part of the event structure's covering graph. From a global state, only minimal actions with respect to an order relation on events can be executed. This order relation can be $\longrightarrow$, for the asynchronous execution and $\longrightarrow \cup \rightsquigarrow$ for the conflict dependent and consensus execution models. In order to make sure that an event is minimal for the hypergraph, it must not be a vertex of an hyperarc. From theorem 1, we know that a single unfolding of hyperarcs containing minimal events ensures that those minimal events are not contained in any hyperarc anymore. Unfolding can create new minimal events, so the calculus of a valid state is an iteration. This iteration stops when there are no more minimal events contained in an hyperarc (in such a case, a valid state was found), or when an unfolding obliges to use a rule twice (in such a case, we can't find a valid state, an though we can not continue the execution of our specification).

An unfolding of an hyperarc h in an hypergraph $G = (T(G), H(G))$ is noted $unfold(G, h)$, and is the hypergraph G' such that $\exists (h, R) \in \mathcal{G} \wedge G \underset{\mathcal{G},(h,R)}{\longrightarrow} G'$ (as our graph grammars are deterministic, (h, R) is unique, and so is G'). A simple algorithm for the calculus of a valid state from an hypergraph G is now defined. Let us call $G_\emptyset = (\emptyset, \emptyset)$ the empty hypergraph, containing no events and no hyperarcs.

```
procedure valid-state(G) =
     G' := G;
     R := {h ∈ H(G)|∃e ∈ min(G, -->) ∧ e ∈ h};
     while R≠ ∅∧G' ≠ G∅ do
          choose h from R; R := R-{h};
          G' := develop-hyperarc(G',{h},h);
     done;
     return G';
end;
procedure develop-hyperarc(G,P,h) =
     G':= Unfold(G,h);
     R :={h ∈ H(G')|∃e ∈ min(G'−G, -->) ∧ e ∈ h};
     while R≠ ∅∧G' ≠ G∅ do
          choose h from R; R := R−{h};
          if h ∈ P then G' := G∅ else G':= develop-hyperarc(G',P∪{h},h);
     done;
     return G';
end
```

Our grammar is proper, which means that no vertex is deleted from a state by rewriting. As our grammar is also separated, there is no intersection between hyperarcs, so developing an hyperarc won't connect edges to the vertices of other hyperarcs in $H(G)$. Consequently, every hyperarc of $H(G)$ can be developed separately.

4.2. Transitions

Once a valid global state G has been calculated, an execution step can be performed. An event e is *executable* if all its predecessors have been executed, and if no event e' such that $e \sharp e'$ have been executed. The execution of an event e from a state G leads to the minimal valid state G', in which e and all events conflicting with e have been removed. This ensures that an execution will never contain conflicting events.

Definition 3:

Let us note $\downarrow (e, G)$ the set containing an event e and its successors in G.

$$\downarrow (e,G) = e \cup \{ \quad e' \in E | \exists e_0, e_1, .., e_n \in E^n, e_0 = e \\ \wedge e_n = e' \wedge \forall 1 < i < n (e_i \longrightarrow e_{i+1} \vee e_i \rightsquigarrow e_{i+1})\}$$

Definition 4:

Let us note $G_{|E'}$ the restriction of an hypergraph G to a set E'. For an hypergraph $G = \big(< E, \longrightarrow, \rightsquigarrow, \sharp >, H(G)\big)$, $G_{|E'} = \big(< E \cap E', \longrightarrow', \rightsquigarrow', \sharp' >, H'(G)\big)$, where:

- $\longrightarrow' = \{(e, e') \in \longrightarrow | e \in E' \wedge e' \in E'\}$,
- $\rightsquigarrow' = \{(e, e') \in \rightsquigarrow | e \in E' \wedge e' \in E'\}$,
- $\sharp' = \{(e, e') \in \sharp | e \in E' \wedge e' \in E'\}$,
- $H'(G) = \{h \in H(G) | \forall e \in h, e \in E'\}$.

4.3. Asynchronous execution of a MSC

An HMSC might be executed in an *asynchronous* mode, which imposes no constraint on conflicts. An event is executable if it is minimal for the order relation ($\longrightarrow$). Note that this mode implements the meaning of choice recommended by norm Z.120. For any valid state $G = \big(< E, \longrightarrow, \rightsquigarrow, \sharp >, H(G)\big)$, we have:

$$\frac{e \in min(G, \longrightarrow) \wedge \nexists h \in H(G) \mid e \in h}{G \xrightarrow{e} G'}$$, where G'=`Next-State-Asynchronous`(G, e)

```
procedure Next-State-Asynchronous(G,e)
  G':=G;
  R:= {h ∈ H(G')|∃e'♯e ∧ ∃e'' ∈ h ∧ e'' ∈↓ (e',G)};
  while R≠ ∅ do
    choose h from R; R:= R−{h};
    if ∃e'' ∈ h|∀e♯e', e'' ∉↓ (e',G') then G':= unfold(G',h);
    else G':= (T(G'), H(G') − {h});
    R:= {h ∈ H(G')|∃e'♯e ∧ ∃e'' ∈ h ∧ e'' ∈↓ (e',G')};
  done;
  G':= G'|E'−({e}∪{↓(e',G')|e'♯e});
  Return(valid-state(G'));
end
```

First, any hyperarc $h \in H(G)$ that contain both conflicting and non-conflicting events wrt e is unfolded. The unfoldings stop when any hyperarc of G contains only conflicting events (in that case they will be removed by the restriction on $E - (\{e\} \cup \{\downarrow e' | e' \sharp e\})$, or non-conflicting events (these hyperarcs will be preserved by the restriction). The hypergraph G' obtained is restricted to the set of executable events, and the next valid state reachable from $G'_{|E'-(\{e\}\cup\{\downarrow e' | e' \sharp e\})}$ is calculated.

An HMSC is not always executable in asynchronous mode, and simulation may lead to a deadlock state. Let us consider HMSC H in Figure 7, and its normalized grammar in Figure 8 . Simulating H in an asynchronous mode is impossible. As event a is the only event performed on instance A, and as the event structure representation contains multiple copies of this event that are all minimal events for $\longrightarrow$, it is impossible to find a finite unfolding of the graph grammar where an event labeled by a would be minimal and not contained in any hyperarc. Consequently, the start state for an asynchronous execution of the grammar of Figure 8 is $G_\emptyset$.

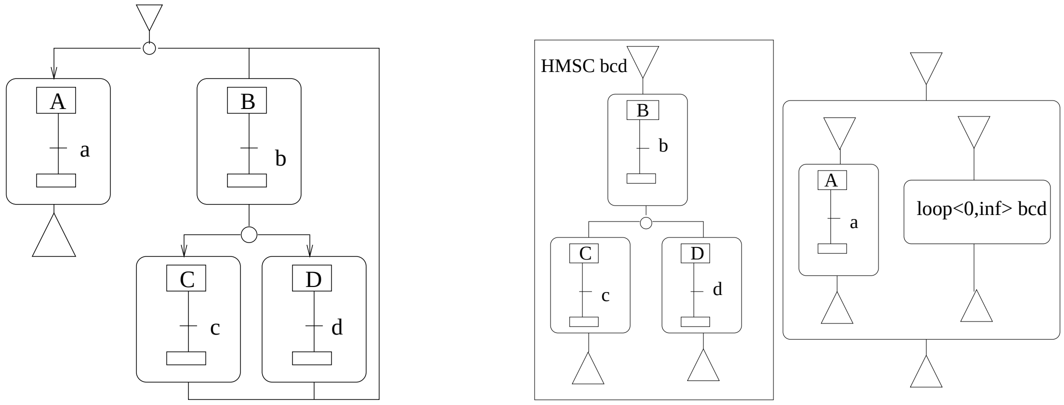

Figure 7. a) HMSC H. b)HMSC H'.

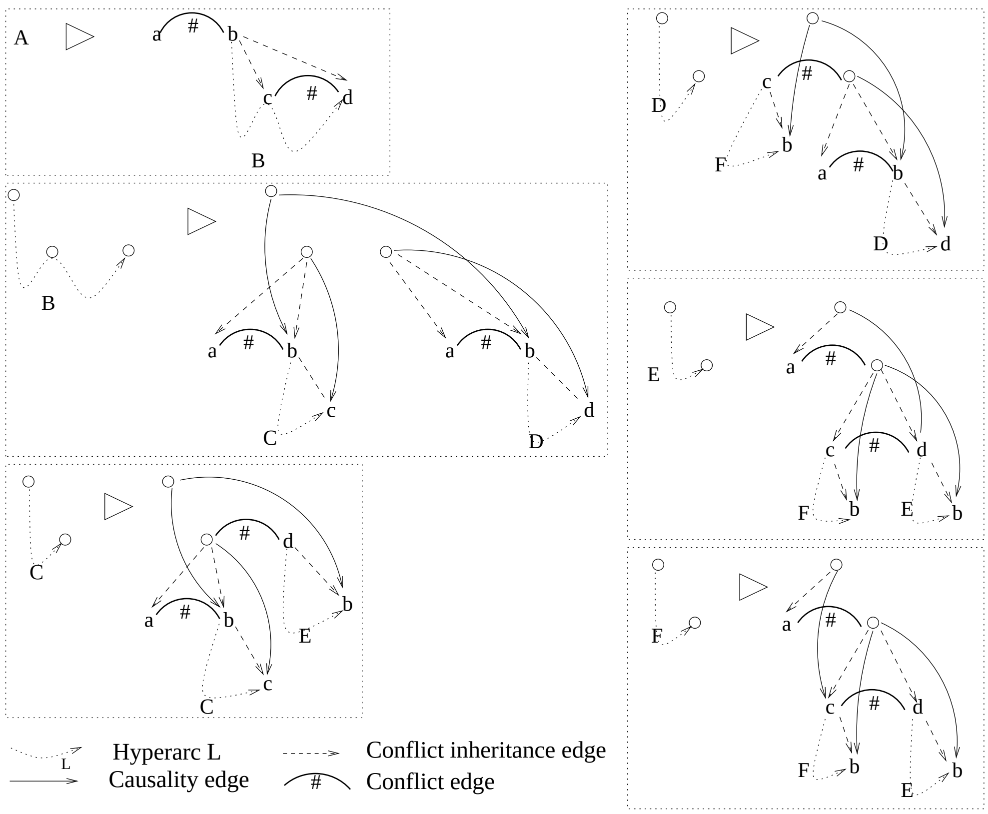

Figure 8. An example of grammar normal form calculated from HMSC in Figure 7.

When an HMSC can not be executed in asynchronous mode, it may be treated by finding an equivalent HMSC (H and H' in Figure 7 have the same POF semantics). Even if scenarios such as H seem to be ill-formed, they express interesting languages and must not be ignored. The language recognized by H is $L = \{w \in A^* || w|_a = 1 \wedge |w|_b = |w|_c + |w|_d\}$. Such an HMSC could be used for defining properties (for example expressing the fact that the number of connections and deconnections in a system must be the same). Most often, the admitted behaviour of the HMSC of Figure 7-a is that a is an exit event for the loop structure. So, the system can perform $b.c$ or $b.d$ an unlimited number of times, until an event a occurs. Within this interpretation, a conflict can be seen as a new kind of synchronization.

4.4. Conflict dependent executions

Conflict dependent executions consider conflicts as causality barriers. If $e \rightsquigarrow e'$, then e' can not be executed as long as e is involved in a conflict. So, once a conflict has been solved, any minimal event in the chosen scenario can be executed. The calculus of a valid state will be done with $\dashrightarrow = \longrightarrow \cup \rightsquigarrow$.

$$\frac{e \in min(G, \longrightarrow \cup \rightsquigarrow) \wedge \nexists h \in H(G) \mid e \in h}{G \xrightarrow{e} G'} \quad \text{with } G' = \text{Next-State}(G, e)$$

The calculus of the next valid state reached after executing an event e is different from the asynchronous case: the conflict inheritance relation does not have the same meaning, and have to be updated after each transition.

```
procedure Next-State(G,e)
  G':=G;
  R:= {h ∈ H(G')|∃e'♯e ∧ ∃e'' ∈ h ∧ e'' ∈↓ (e',G)};
  while R ≠ ∅ do
    choose h from R; R := R- {h};
   if ∃e'' ∈ h|∀e♯e', e'' ∉↓ (e',G') then G':= unfold(G',h);
    else G':= (T(G'), H(G') − {h});
    R:= {h ∈ H(G')|∃e'♯e ∧ ∃e'' ∈ h ∧ e'' ∈↓ (e',G')};
  done;
  G':= G'|E'−({e}∪↓(e',G')|e'♯e});
   ⇝':= {(e', e'') ∈⇝' |∃e''' ∈ E' ∧ e'♯e'''};
  Return(valid-state(G'));
end
```

When conflicts are removed, any event connected to a formerly conflicting event by $\rightsquigarrow$ becomes a minimal event.

The valid executions of the example of Figure 7 are words from $L = \{b.c + b.d\}^*.a$.

4.5. Consensus executions

The *consensus* execution mode authorizes an instance to choose a scenario only when all the instances involved in the decision can perform the same choice. This kind of execution requires a consensus between participating instances. An instance can not continue at a choice as long as an agreement is not found. Conflicts act as synchronization barriers.

$$\frac{e \in min(G, \longrightarrow \cup \rightsquigarrow) \wedge \nexists H \in \mathcal{H} \mid e \in H \wedge \forall e' | e' \sharp e, e' \in min(G, \longrightarrow \cup \rightsquigarrow)}{G \stackrel{e}{\longrightarrow} G'}$$

with $G' =$`Next-State`(G, e)
The calculus of the next valid state is the same that for a conflict dependent execution.

5. Example

Let us consider the simple data transfer protocol defined on Figure 9. First, $User_A$ sends a connection request to $Network_B$. This connection can be accepted or rejected. Once it is accepted, $User_A$ can send data messages to $Network_B$, or close the connection. $Network_B$ forwards received data to $User_B$.

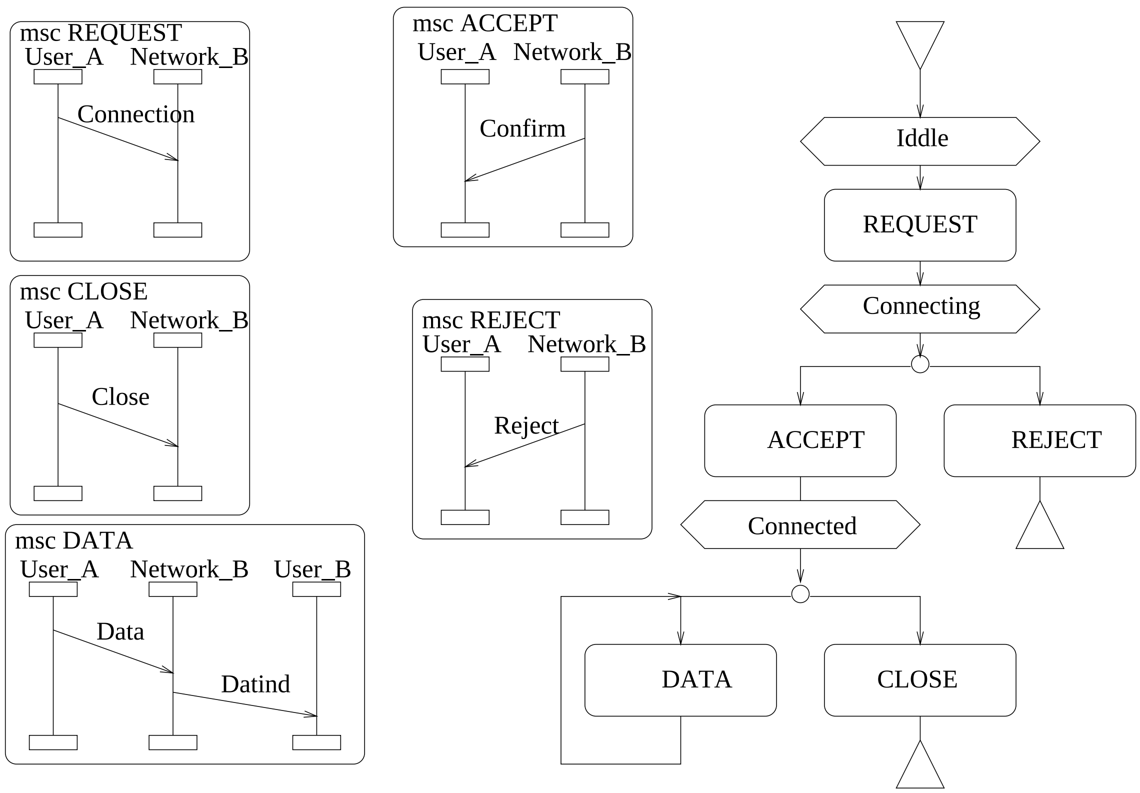

Figure 9. A simple example protocol.

The covering graph of the event structure representation of the HMSC in Figure 9 is represented in Figure 10. We indicated the distance for each event on the graph by drawing frontiers between distant events. One can easily note that the events contained between two frontiers are also events appearing in the same rewriting rule. One can also note that at distance 6, the set of events between two frontiers becomes a regular pattern (it implies that rule F of the normalized graph grammar in Figure 11 can be continued by developing F another time).

Figure 12 shows the different states reached during the asynchronous execution of $w =!connection.?connection.!confirm.?confirm.!data.!data.!data$. The size of the graph

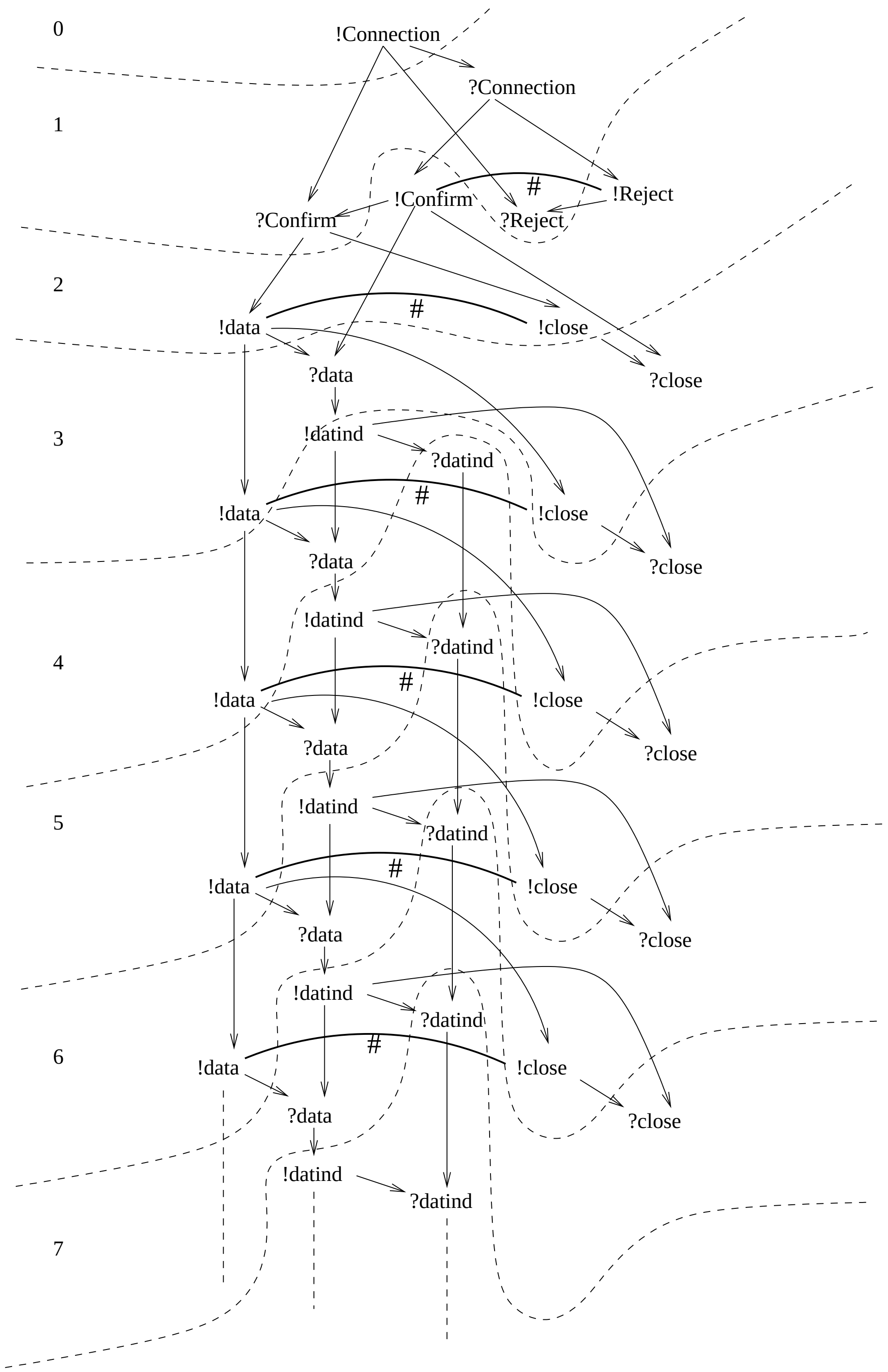

Figure 10. Covering graph for HMSC in Figure 9.

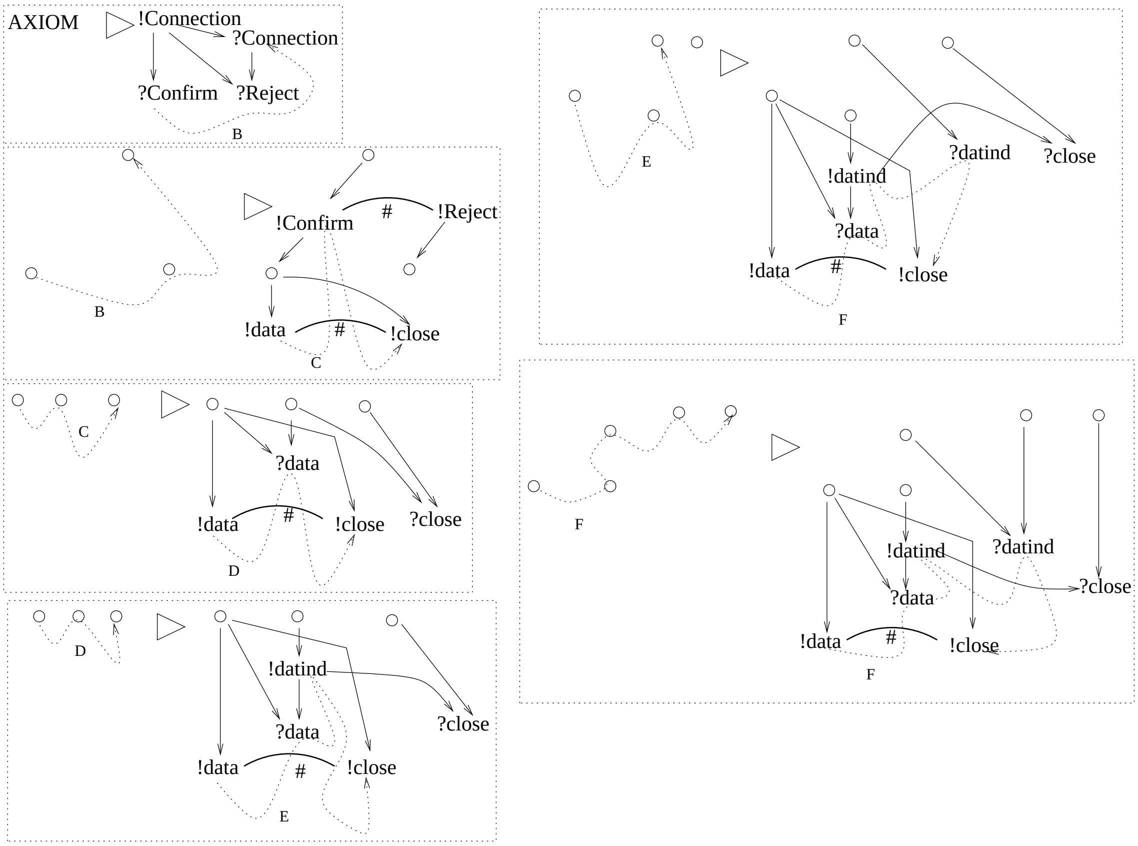

Figure 11. Normalized grammar for HMSC in Figure 9.

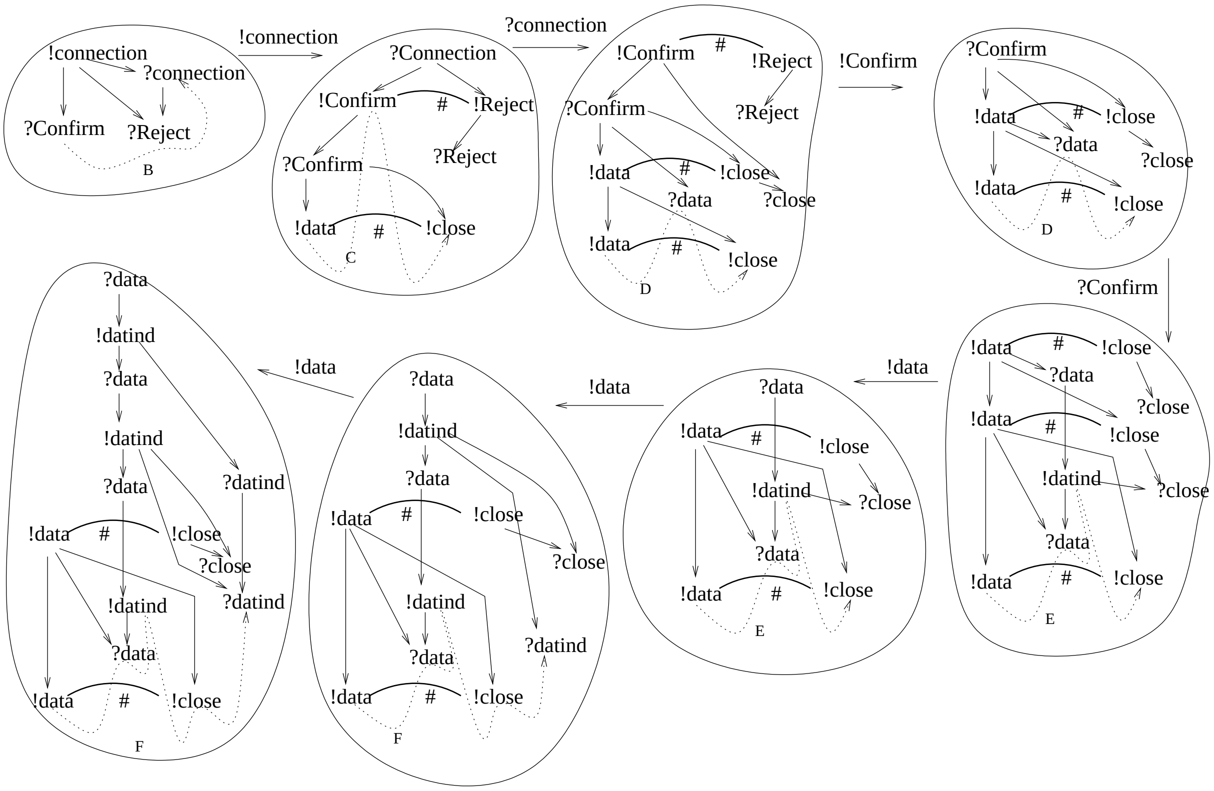

Figure 12. Execution of grammar in Figure 11.

grows in a regular way. This denotes a process divergence between instance $User_A$ and the set of instances $\{Network_B, User_B\}$.

6. Conclusion and perspectives

We have introduced a framework for simulation of scenarios based on graph grammars. Such executions can help detecting undesired behaviours at an early stage of a specification. In addition to providing help in the conception of distributed systems, graph grammars also give useful information about the synchronization of a specification.

Let us consider the communication graph of an HMSC P, $CG(P) = (I, \longrightarrow_P)$ such that $i \longrightarrow_P j$ when instance $i \in I$ can send a message to instance $j \in I$ in P. From this graph we can partition the set of instances into subsets of communicating instances, ie the connected components of the graph $CG(P)$.

From the previous examples, we can see that the size of the states can grow very fast if a set of communicating instances is few synchronized. On the contrary, the size of the states may remain bounded if all the sets of communicating instances are strongly synchronized. This let us think that graph grammar representation could be used to define concurrency measures of HMSCs.

REFERENCES

1. Ben-Abdallah.H,Leue.S Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts in: E. Brinksma (ed.), Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97, Enschede, The Netherlands, April 1997, Lecture Notes in Computer Science, Vol. 1217, p. 259-274, Springer-Verlag, 1997.
2. Caucal.D. On the regular structure of prefix rewriting. *Theoretical Computer Science*, (106):61–86, 1992.
3. Habel.A. Hyperedge replacement: grammars and graphs. *Lecture Notes in Computer Science*, (643), 1989.
4. Helouet.L, Jard.C.,Caillaud.B. An effective equivalence for sets of scenarios represented by Message Sequence Charts. INRIA research report no 3499, ftp://ftp.inria.fr/INRIA/publication/RR/RR-3499.ps.gz
5. Heymer.S A non-interleaving semantics for MSC. *SAM98:1st conference on SDL and MSC*,281-290, 1998.
6. Katoen.J.P, Lambert.L, Pomsets for message sequence charts. *SAM98:1st conference on SDL and MSC*,281-290, 1998.
7. Reniers.A, Mauw.S. High-level message sequence charts. Technical report, Heindoven University of Technology, 1996.
8. Winskel.G, Nielsen.M, Plotkin.G. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13, 1981.
9. Graubmann.P, Rudolph.E, Grabowski.J. Tutorial on message sequence charts (msc'96). FORTE/PSTV'96, october 1996.
10. ITU-T Message Sequence Chart (MSC) *ITU-T Recommendation Z120*, October 1996.