Electronic Notes in Theoretical Computer Science 133 (2005) 193–215 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Merging Scenarios

# Jacques Klein[1](#_bookmark0) ,[2](#_bookmark0) Benoit Caillaud[3](#_bookmark0) Lo¨ıc H´elou¨et[4](#_bookmark0)

*IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France*

Abstract

This paper proposes a merge operator for behavioral requirements expressed by Message Sequence Charts and shows how this product can be systematically used to integrate new behaviors in an existing one. First the merge operator is defined as a fibered product of scenario descriptions. This product is then used to integrate a consensus mechanism to solve the non-local choice problem.

*Keywords:* Scenarios, distributed systems, modeling, composition.

# Introduction

Scenario languages define typical executions of systems. They are used to rep- resent output traces, but also to capture requirements of distributed systems. Even if several dialects exist, all scenario languages are based on a similar idea: they depict systems runs as compositions of partially ordered sets of events. A drawback of scenarios is that the number of participants in a given interaction cannot be parameterized. With a fixed number of objects in an interaction, it is hard to describe behaviors of systems with dynamic architecture. Message Sequence Charts propose instance creation, but this mechanism needs to know a priori the name of instances that will be created.

Another drawback of scenarios is that a description of a system is usually defined by a set of scenarios, which represent typical uses of the system under

1 Email: [jacques.klein@irisa.fr](mailto:jacques.klein@irisa.fr)

2 (This work has been partially supported by Families Project, ITEA project ip02009)

3 Email: [benoit.caillaud@irisa.fr](mailto:benoit.caillaud@irisa.fr)

4 Email: [loic.helouet@irisa.fr](mailto:loic.helouet@irisa.fr)

1571-0661 © 2005 Elsevier B.V. Open access under C[C BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2004.08.065

design in a given situation, but comport some redundancies. The intended behavior of a system can be seen as a combination of all these views. So far, no satisfactory merge operator exists. Scenario languages are all equipped with alternative, sequential, or parallel composition, but these operators do not capture the notion of redundancy that may exist between the composed views.

A solution proposed is to gather coherently redundant views given as Live Sequence charts [[8](#_bookmark28)]. The main idea is to combine scenarios at runtime. LSCs are executed in parallel, and events that can be executed in several views are synchronized when possible. From an initial set of live scenarios, an event is executed, and new scenarios are “triggered” by this event execution. Two scenarios are declared inconsistent if they contradict each other on the order of common events. The main drawback of this approach is that inconsistency between scenarios may not be discovered if the simulations performed do not pass through a faulty configuration.

This paper addresses a second approach that consists in the construction of a new model, preferably using the same scenario language, which is the smallest model to contain all the views composed. Such a model does not always exist, and in some cases it is not unique. However, when common parts in scenarios are clearly identified, this model can be computed. The language chosen is Message Sequence Charts, a scenario language standardized by ITU [[11](#_bookmark29)], but our approach can be adapted to any partial order based language. The merge framework proposed uses a fibered product of scenarios. This product first identifies pairs of scenarios that must be “synchronized” in two HMSCs, and realizes their union with an amalgamated sum. We then show the usefulness of this construction on a concrete application, ie, the introduction of a consensus algorithm to localize choices in a HMSC.

The benefits of scenario merging do not only concern view composition. In fact, an endogenous merge operator can be used to propose formal and well founded model transformations and design patterns for scenarios. The paper is organized as follows: Section [2](#_bookmark1) recalls some basic notions on Message Sequence Charts. Section [3](#_bookmark4) proposes a definition of amalgamated sum. Section [4](#_bookmark7) defines the complete fibered product. Section [5](#_bookmark12) shows an application of fibered product to eliminate non-local choices from HMSCs, and section [6](#_bookmark20) concludes this work.

# Message Sequence Charts

Message Sequence Charts (or MSC for short) is a scenario language standard- ized by ITU [[11](#_bookmark29)]. MSCs are composition of very simple chronograms by means

of sequence, alternative, and iteration operators. MSCs propose two specifica- tion levels. At the lowest level, basic Message Sequence Charts (or bMSCs for short) describe simple communication patterns between entities of the system called *instances*. These chronograms are then composed by several levels of High-level Message Sequence Charts (or HMSC for short), a kind of bMSC automaton.

In a bMSC, instances are represented by a vertical axis. Message exchanges are represented by arrows labeled by message names from the emitting to the receiving instance, and communications are supposed to be asynchronous. A bMSC defines a set of events, which are occurrences of actions in the system (message emissions, receptions, atomic actions or operations on timers), and a precedence relation on these events: a message emission must precede the corresponding reception, and events are totally ordered along instance axis (excepted in specific parts of the axis called coregions). Process algebra se- mantics for bMSCs have been proposed [[16](#_bookmark36)], but it seems rather natural to give bMSCs a non interleaving semantics as in [[10](#_bookmark30)] and [[12](#_bookmark31)]. In the sequel, our formal definition of bMSC will be based on the notions of preorders and partial orders. A *preorder* on a set of elements *E* is a relation *R* ⊆ *E*2 that is reflexive (ie. ∀*e* ∈ *E, eRe*) and transitive (ie. ∀*e, f, g* ∈ *E, eRf* ∧ *f Rg* =⇒ *eRg*). A *par- tial order* is an antisymmetric preorder (ie. ∀*a, b* ∈ *E, aRb* ∧ *bRa* =⇒ *a* = *b*). As already mentioned, bMSCs define a causal order between message emis- sions and receptions, and a partial ordering on events situated on the same

instance. They can then be formally defined as follows:

Definition 2.1 (Basic Message Sequence Charts) *Let I be a ﬁnite set of instances. A* bMSC *over I is a tuple M* = (*E,* ≤*, A, α, φ,* ≺)*, where E is a set of events,* ≤ *is a preorder on E, A is a set of actions, α* : *E* → *A maps events to actions, φ* : *E* → *I maps events to instances,* ≺⊆ *E* × *E is a partial bijection pairing message emissions and receptions, such that* ≺⊆≤*. When no instance set is speciﬁed, a bMSC is a tuple M* = (*I, E,* ≤*, A, α, φ,* ≺)*, where I is a ﬁnite instance set and M* = (*E,* ≤*, A, α, φ,* ≺) *is a bMSC over I.*

A bMSC *M* will be called *well-formed* whenever ≤ is a partial order re- lation. For a bMSC *M* = (*E,* ≤*, A, α, φ,* ≺), we will denote by *Min*(*M*) =

{*e* ∈ *E*|∀*e*' ∈ *E, e*' ≤ *e* ⇒ *e*' = *e*} the set of minimal events for the preorder relation, i.e the set of events that have no causal predecessor. Figure [1](#_bookmark2) shows an example of bMSC, with 3 instances (*sender*, *medium* and *receiver*), that exchange several messages (*data*, *inf o* and *ack*). The events of this bMSC are the emission and receptions of messages , and an atomic action *action* executed by the *sender*. Note that events *e*5 and *e*6, that symbolize emissions of messages *inf o* and *ack* are situated in a coregion, ie no order is imposed between these two events.

bMSCs alone do not have a sufficient expressive power: they can only de- fine finite behaviors, without real alternatives (the only alternatives in the behaviors depicted by a bMSC are due to possible interleavings). For this reason, MSCs have been extended with higher-level constructs, namely HM- SCs [[18](#_bookmark38)]. Roughly speaking, HMSCs are a kind of transition systems labeled by bMSCs.

Definition 2.2 (Labeled Transition Systems) *A* labeled transition sys- tem *(or LTS for short) is a tuple* S = (*S, s,* Σ*,T* )*, where S is a set of states, s* ∈ *S is the initial state of* S*,* Σ *is a ﬁnite alphabet, and T* ⊆ *S* × Σ × *S is a set of transitions. In the sequel, we will denote by α*(*t*) = *a the* label *of a transition t* = (*p, a, q*) ∈ *T. We will write p* −*a*→ *q whenever* (*p, a, q*) ∈ *T. State p will be called the* origin *of t, denoted α*−(*t*)*, and q the* goal *of t, denoted α*+(*t*)*.*

^

^

Definition 2.3 (High level Message Sequence Charts) *Let I be a ﬁnite set of instances. A* HMSC *over I is a tuple H* = (S*,* M*, λ*) *where:* S = (*S, s,* Σ*,T* ) *is a labeled transition system called the* support automaton *of H,* M *is a ﬁnite set of bMSCs over I, λ* : *T* → M *maps transitions to bMSCs. When no instance set is speciﬁed, HMSCs are deﬁned as quadruples H* = (*I,* S*,* M*, λ*)*, where I is a ﬁnite instance set and* (S*,* M*, λ*) *is a HMSC over I.*

^

An example of HMSC is given Figure [2](#_bookmark3). The notion of sequential composi- tion (noted •) is central to understand HMSCs. Roughly speaking, sequential composition of two bMSCs consists in gluing both diagrams along their com- mon instance axes. Note that this sequence does only impose precedence on events situated on the same instance, but that events situated on different in- stance in two bMSCs *M*1 and *M*2 can be concurrent in *M*1 • *M*2. Sequential composition can be formally defined as follows:

Definition 2.4 (Sequential Composition) *The* sequential composition *of two bMSCs M*1 *and M*2 *is the bMSC M*1 • *M*2 = (*E*1 *E*2*,* ≤1•2*, α*1 ∪ *α*2*, φ*1 ∪

*φ ,A* ∪ *A ,* ≺ ≺ )*, where:*≤ = ≤ ≤ {(*e, e*') ∈ *E* × *E* |*φ* (*e* ) =

2

1

2

1

2

1•2

1

2

1

2

1

1

*φ* (*e* )} ∗

2 2

As already mentioned, HMSCs are a kind of automaton labeled by partial

orders. However, the automaton contained in a HMSC is only a support for sequential composition, and should not be considered as a synchronization defining a mandatory global state for all the instances. For this reason, the states of the support automaton in a HMSC *H* will often be called the *nodes* of *H*.

A *path* in a HMSC is a sequence of transitions *T* = *t*1*.t*2 *tk*, such that

the origin of *ti*+1 is equal to the goal of *ti*: For every *i* = 1 *k* − 1*, α*+(*ti*)=

*α*−(*ti*+1). Using sequential composition, each finite path *T* = *t*1*.t*2*.... tk* of a

HMSC defines a bMSC *OT* = *λ*(*t*1) • *λ*(*t*2) • ··· • *λ*(*tk*). A path *T* = *t*1*.t*2 *tk*

is a *circuit* if the origin of *t*1 is equal to the goal of *tk*. An *acyclic path* in a HMSC is a path *T* = *t*1*.... tk* which contains no cycle, that is: ∀*i* ≤ *j*, *α*−(*ti*) /= *α*+(*tj*). A *maximal acyclic path* of H is an acyclic path *T* = *t*1 *tk*

such that any extension of path *T* by one transition contains a cycle.

A *choice node* in a HMSC is a node that is the origin of two or more distinct transitions. As transitions in HMSCs are labeled by partial orders, choices can depict situations where several instances can decide to perform a scenario or another. This situation is called *non-local choice*. It was first identified in [[14](#_bookmark34),[2](#_bookmark21)], and then refined in [[9](#_bookmark32)]. Consider, for instance the choice node in HMSC Figure [2](#_bookmark3). Following the description given by scenarios *M*1 and *M*2, from this node instance *A* can decide to send message *m*1 and then *B* must conform to this choice and receive *m*1, or instance *B* can decide to send the message *m*2, in which case instance *A* should receive *m*2. However, an im- plementation of such description without additional communications between

*A* and *B* would probably lead to a situation where *A* sends message *m*1 while

*B* sends message *m*2, and both instances are then deadlocked.

bMSC example

sender

medium

receiver

e1

data

e4

e5

info

e2

e7

ack

e6

e3

action

Fig. 1. An example of bMSC

HMSC H1

bMSC M0

A

B

m0

bMSC M1 bMSC M2

M0

M1

M2

A

B

m1

A

B

m2

Fig. 2. non-local choice HMSC

The common understanding of choices in HMSCs is that the first instance able to perform a choice selects a behavior. The following instances reaching the same occurrence of this choice have to conform to the chosen scenario. So, the MSC semantics assumes an implicit agreement between instances (e.g., one instance decides and communicates its decision to all other instances, while

the others wait until they are notified of the decision). When HMSCs are used to define a set of behaviors at a high abstraction level, this type of specifi- cation is not shocking: the only behaviors allowed are the scenarios depicted by each branch. However, when HMSCs are supposed to be precise enough to be implemented, non-local choices can have several conflicting interpreta- tions. Indeed, the description of figure [2](#_bookmark3) can have several meanings. The first interpretation is that scenarios *M*1 and *M*2 are the only possible behaviors of the system. Communications must be added to the model to avoid message crossings. Another possible interpretation is that a third scenario where *m*1 intersects *m*2 is possible. This scenario should appear in the original HMSC. The first definition of non-local choices in [[2](#_bookmark21)] assumes that any instance should communicate with other instances on each branch of a choice. This assumption limits the search for non-local choice to the set of edges leaving choice nodes. However, when considering weak sequential composition of bM- SCs with disjoint set of instances, non-local choice is not a local property of choice nodes, but must be verified on the complete support automata [[9](#_bookmark32)]. In

the sequel, we will adopt the following definition of non-local choice:

Definition 2.5 (Local Choice) *Let c be a choice node. c is* local *if and* *only if* ∃*i* ∈ *I such that* ∀*p, path of H starting from c, φ*(*Min*(*Op*)) = {*i*}*. i will be called the* deciding instance *of choice c.*

Notice that the local choice property can be checked by considering maxi- mal acyclic paths only. It is therefore decidable on finite HMSCs. Section [5](#_bookmark12), shows how a consensus algorithm can be inserted automatically to transform a non-local HMSC into a local one.

# Amalgamated Sum of bMSCs

So far, bMSC composition is limited to parallel or sequential composition, iteration or choice. Other operations on bMSCs have been proposed, such as instance refinement [[15](#_bookmark35)], message refinement [[5](#_bookmark25)], virtuality [[17](#_bookmark37)], or more recently projections [[6](#_bookmark26)]. However, when two bMSCs depict different viewpoints of the same behavior, one feel the need for a merge operation that would glue the two scenarios to produce a result that contains both operands without creating copies of similar elements. This operator cannot be expressed by means of sequential nor parallel composition. We propose a merge operator for bMSCs called *amalgamated sum*. This amalgamated sum uses concepts of category theory. However, for the sake of conciseness, only what is strictly necessary has been included in the paper. More details on this topic can be found in [[4](#_bookmark24),[3](#_bookmark23)]. First, we need to define the notion of bMSC morphisms, that will be essential to define common parts in scenarios.

Definition 3.1 (bMSC Morphism) *An* instance set morphism *is an injec- tive mapping µ* : *I* −→ *I*' *from an instance set I to another instance set I*'*. Let*

*I and I*' *be two ﬁnite sets of instances and µ* : *I* → *I*' *an instance set mor-*

0

*phism. A* bMSC morphism *along µ*0*, from M* = (*E,* ≤*, A, α, φ,* ≺)*, a bMSC*

*over I, to M*' = (*E*'*,* ≤'*, A*'*, α*'*, φ*'*,* ≺')*, bMSC over I*'*, is a pair of mappings*

*µ* =*< µ ,µ > where µ* : *E* → *E*' *is injective, µ*

: *A* → *A*' *is a renaming*

1 2 1 2

*mapping, and:*

1. ∀(*e, f* ) ∈ *E*2*,e* ≤ *f* ⇒ *µ* (*e*) ≤' *µ* (*f* )

1 1

1. ∀(*e, f* ) ∈ *E*2*,e* ≺ *f* ⇒ *µ* (*e*) ≺' *µ* (*f* )

1 1

1. *µ* ◦ *φ* = *φ*' ◦ *µ*

0 1

1. *µ* ◦ *α* = *α*' ◦ *µ*

2 1

*When no instance set morphism is speciﬁed, bMSC morphisms are deﬁned by triples µ* = (*µ*0*, µ*1*, µ*2) *such that (µ*1*, µ*2*) is a bMSC morphism along µ*0*.*

Note that property (*iii*) also means that all events located on a single

instance of *M* are sent by *µ*1

on a single instance of *M*'.

Definition 3.2 (Amalgamated Sum of Two Sets) *Let I, J and K be three ﬁnite sets. Let f* : *I* → *J and g* : *I* → *K be two injective maps. The amalga- mated sum J f* +*g K is deﬁned as J f* +*g K* = *J*\*f* (*I*) *K*\*g*(*I*) *I. The amalgamated sum yields two injections f*˜ : *J* → *J f* +*g K and g*˜ : *K* → *J f* +*g K deﬁned as follows:*

⎧⎨ ∀*i* ∈ *f* (*I*)*, f*˜(*i*)= *f* −1(*i*) ⎧⎨ ∀*i* ∈ *g*(*I*)*, g*˜(*i*)= *g*−1(*i*)

⎩ ∀*i* ∈ *J* \ *f* (*I*)*, f*˜(*i*)= *i* ⎩ ∀*i* ∈ *K* \ *g*(*I*)*, g*˜(*i*)= *i*

Note that as we use (disjoint union) in our definition, the result of an amalgamated sum can contain several copies of similar elements.

Amalgamated sums of sets will be used to amalgamate sets of instances, events or actions of two bMSCs to be composed.

Definition 3.3 (Amalgamated Sum of two bMSCs) *Let M*0 = (*I*0*, E*0*,*≤0

*, A*0*, α*0*, φ*0*,* ≺0)*, M*1 = (*I*1*, E*1*,* ≤1*, A*1*, α*1*, φ*1*,* ≺1)*, M*2= (*I*2*, E*2*,* ≤2*, A*2*, α*2*, φ*2*,*≺2

) *be three bMSCs and f* =*< f*0*, f*1*, f*2 *>*: *M*0 → *M*1*, g* =*< g*0*, g*1*, g*2 *>*: *M*0 → *M*2 *be two bMSCs morphisms. The amalgamated sum of M*1 *and M*2 *wrt. f and g is the bMSC M* = *M*1 *f* +*g M*2 *where M* = (*I, E,* ≤*, A, α, φ,* ≺) *is deﬁned by:*

* *I* = *I*1 *f* +*g I*2*; E* = *E*1 *f* +*g E*2 *; A* = *A*1 *f* +*g A*2*;*

0 0 1 1 2 2

* *Preorder relation* ≤ *is the transitive closure of f*˜1(≤1) ∪ *g*˜1(≤2)*;*

⎧⎪*α*1(*e*) *if e* ∈ *E*1\*f*1(*E*0) ⎧⎪*φ*1(*e*) *if e* ∈ *E*1\*f*1(*E*0)

* ∀*e* ∈ *E, α*(*e*)= ⎪⎨*α*2(*e*) *if e* ∈ *E*2\*f*2(*E*0) *, φ*(*e*)= ⎪⎨*φ*2(*e*) *if e* ∈ *E*2\*f*2(*E*0)

⎪⎪⎩*α*0(*e*) *otherwise*

* ≺= *f*˜ (≺ ) ∪ *g*˜ (≺ )*.*

⎪⎪⎩*φ*0(*e*) *otherwise*

1 1 1 2

*The bMSC M*0 *is called the* interface *of the amalgamated sum M*1 *f* +*g M*2*.*

Let us illustrate the use of amalgamated sum on the example of Figure [3](#_bookmark5). Considering bMSCs *M*1 = (*I*1*, E*1*,* ≤1*, A*1*, α*1*, φ*1*,* ≺1) and *M*2 = (*I*2*, E*2*,* ≤2

*, A*2*, α*2*, φ*2*,* ≺2) as two partial observations of the same system, we want to produce a behavior that contains *M*1 and *M*2. Let us also suppose that even if *M*1 and *M*2 have different instance sets, instance *X* in *M*2 and instance *sender* in *M*1 (resp. *Y* and *medium*) represent the same object in the system. Intuitively, merging *M*1 and *M*2 then amounts to inserting an atomic action between *data* emission and *ack* reception in *M*1, and renaming the instances. Formally, the merge consists in the definition of an interface that identifies the common elements (events, action name, and instances) in *M*1 and *M*2 and renames them. For our example, this is done using a new bMSC *M*0 = (*I*0*, E*0*,* ≤0*, A*0*, α*0*, φ*0*,* ≺0), and two bMSC morphisms *f* : *M*0 → *M*1 and *g* : *M*0 → *M*2 defined below.

* Morphism *f* : *M*0 → *M*1 is a triple *f* =*< f*0*, f*1*, f*2 *>*, where:
  + *f*0 : *I*0 → *I*1 is the identity,
  + *f*1 : *E*0 → *E*1 sends respectively *eI*1*, eI*2*, eI*3*, eI*4 onto *ev*1*, ev*2*, ev*3*, ev*4,
  + *f*2 : *A*0 → *A*1 is the identity.
* Morphism *g* : *M*0 → *M*2 is a triple *g* =*< g*0*, g*1*, g*2 *>*, where:
  + *g*0 : *I*0 → *I*2 sends *sender* onto *X* and *medium* onto *Y* ,
  + *g*1 : *E*0 → *E*2 sends respectively *eI*1*, eI*2*, eI*3*, eI*4 onto *evt*1*, evt*3*, evt*4*, evt*5,
  + *g*2 : *A*0 → *A*2 sends respectively !*data,* ?*data,* !*ack,* ?*ack* onto !*m*1*,* ?*m*1*,* !*m*2*,*

?*m*2.

The result of the amalgamated sum *M*1 *f* +*g M*2 is the bMSC of Figure [1](#_bookmark2). Note that the names of the resulting instances on common parts are defined by the instance names of the interface (we could have proposed an instance name “*X* + *Sender*” instead of keeping *Sender* in the amalgamated sum.)

Note also that an amalgamated sum of two well-formed bMSCs is not al- ways a well-formed bMSC, as the least preorder containing ≤1 and ≤2 may not be antisymmetric. Consider the example of Figure [4](#_bookmark6). MSC *M*1 imposes

*M*1 *M*2

X

Y

evt1 m1

evt4

evt2

m2 evt5

evt3

action

sender medium

receiver

ev1 data

ev3

ev4

info

ev6

ack

ev2

ev5

*f*

*M*0

sender medium

*g*

eI1

data

eI3

eI2

eI4

ack

Fig. 3. An example of amalgamated sum

*M*1 *f* +*g M*2

A

B

m2 m1

a b

*M*1 *M*2

m2

*eb*

*e*'

*a*

A

B *M*0

a b

A

B

a

A

B

b

*ea*

*e*'

a

m1

*b*

b

Fig. 4. An amalgamated sum that is not well-formed

that *e**a* ≤ *eb* while MSC *M*2

states that *e*' ≤ *e*' . Clearly, if *ea* and *eb* are

respectively identified with *e*'

*a*

and *e*' by the interface, the amalgamated sum

of *M*1 and *M*2 will create a symmetry, which can be easily detected. In such case, the two scenarios are incompatible, at least if one considers a match- ing between events symbolizing actions *a* and *b* in both operands. For more considerations about amalgamated sum properties, consult [[4](#_bookmark24)].

*b*

*a*

*b*

# Fibered product of HMSCs

In many cases, merging bMSCs is not sufficient, and a question that imme- diately arises is how to extend the amalgamated sum approach to HMSCs. The first idea is to work on the set of bMSCs generated by a HMSC. The

composition of two sets of bMSCs would be the set of coherent amalgamated sums obtained by merging pairs of bMSCs from each set. However, the result obtained is not always a set of partial orders that can be generated by a single HMSC. Consider, for example, the two HMSCs of Figure [5](#_bookmark8). Trying to merge the atomic actions labeled by *a* and *b* in both HMSCs would result in the set of bMSCs described by Figure [6](#_bookmark9), where one can say nothing about the ordering relationship between the receipt and sending of the message m, and the events corresponding to the messages labelled by n. Clearly, this set is not generated by a HMSC, as the messages of type *m* can cross an unbounded number of messages of type *n*. This kind of specification can be expressed by means of Compositional Message Sequence Charts [[7](#_bookmark27)], or Extended compositional Message Sequence charts [[13](#_bookmark33)], but not with HMSCs.

bMSC M1

A

a

bMSC M3

B

bMSC M

a

bMSC M2

A

B

m

b

M1

M2

M3

M

b

A

B

n

Fig. 5. Event by event matching

bMSC M1oM3 + M

A

B

a

m

bMSC M1oM2oM3 + M

bMSC M1oM2oM2oM3 + M

b

Fig. 6. Result

A

B

a

n m

n

b

A

B

a

n m

b

We propose a solution based on a partially synchronized product of HMSCs inspired by the fibered product of asynchronous transition systems proposed in [[3](#_bookmark23)]. The fibered product of HMSCs is twofold: First, transitions of the two support automata are synchronized partially. Then, the amalgamated sums of the bMSCs attached to the transitions being synchronized are computed to create new bMSCs. As for the amalgamated sum of section [3](#_bookmark4), the formal definition of the fibered product of HMSCs relies on a notion of morphism.

Definition 4.1 (LTS Morphism) *Let* S1 = (*S*1*, s*1*,* Σ1*, T*1) *and* S2 = (*S*2*, s*2*,* Σ2*, T*2) *be two labeled transition systems (LTS for short). A LTS morphism f* : S1 → S2 *is a pair f* =*< f*1*, f*2 *> where f*1 : *S*1 → *S*2 *is a total function while f*2 : *T*1 *~ T*2 *is a partial function which satisfy:*

^ ^

1. *f*1(*s*1)= *s*2*;*

^ ^

1. *t*1 = (*p, a, q*) *in T*1 *and f*2(*t*1) *deﬁned imply* ∃*b* ∈ Σ2*, f*2(*t*1) =(*f*1(*p*)*, b, f*1(*q*))

*in T*2*;*

1. *t*1 = (*p, a, q*) *in T*1 *and f*2(*t*1) *undeﬁned imply f*1(*p*)= *f*1(*q*) *in* S2*.*

Condition *i*) ensures that morphisms preserve initial states. According to conditions *ii*) and *iii*), any transition *t* ∈ *T*1 of S1 is mapped to a transition of S2 via *f*2 if *f*2 is defined in *t*. In other words, *f*2 defines which transitions of S1 have observable effects in S2. For convenience, we can add an artificial *empty* transition *p* −→ *p* to each state *p* ∈ *S*. With this convention, transitions of S1 that are unobservable in S2 are mapped to empty transition. Hence, we can succinctly rewrite the second and third conditions of definition [4.1](#_bookmark10) as follows:

*є*

*t* = (*p, a, q*) transition of S1

=⇒ ∃*a*' ∈ Σ

∪ *ϵ,*

*f* (*t*)= (*f* (*p*)*, a*'*,f* (*q*)) transition of S

2

2 1 1 2

This convention is used throughout the paper. Therefore, it is now assumed that transition system are completed in every state with an empty transition

of the form *p* −→*є p*.

The partially synchronized product of labeled transition systems was first introduced by Arnold [[1](#_bookmark22)]. It is parameterized by a set of synchronization vectors, which is used to define which pairs of labels must be synchronized. Here, a set of pairs of transitions (instead of labels) is used for this purpose.

Definition 4.2 (Synchronous product) *Let* S1 = (*S*1*, s*1*,* Σ1*, T*1) *and* S2 = (*S*2*, s*2*,* Σ2*, T*2) *be two LTSs. A* synchronization constraint *C is a subset of T*1 × *T*2*. Elements of C are called* synchronization vectors*, and indicate which transitions of* S1 *and* S2 *are synchronized. The* synchronous product *of* S1 *and* S2 *under a synchronization constraint C is the LTS* S = (*S, s,* Σ*,T* ) *where:*

^

^

^

* *S* = *S* × *S ; s* = (*s , s* )*;*

1 2 ^ ^1 ^2

* Σ= (*α*(*t*1)*, α*(*t*2)) | (*t*1*, t*2) ∈ *C*}*;*
* *T* = (*s, σ, s*') ∈ *S* × Σ × *S* | ∃(*t ,t* ) ∈ *C, σ* = (*α*(*t* )*, α*(*t* )) ∧ *s* =

1

2

1

2

(*α*−(*t* )*, α*−(*t* )) ∧ *s*' = (*α* (*t*1)*,α* (*t* ))}*.*

1

2

+

+

2

In the previous section, we have defined a notion of morphism for bMSC. HMSC morphisms can be defined, in a similar way, as triples of morphisms or mappings: i) a morphism of instance sets, ii) a morphism of labeled transition systems and iii) a mapping that associates bMSC morphisms to transitions.

Definition 4.3 (HMSC morphism) *Let H*1 = (*I*1*,* S1*,* M1*, λ*1)*, H*2 = (*I*2*,* S2*,* M2*, λ*2) *be two HMSCs. A HMSC morphism f* : *H*1 → *H*2 *from H*1 *to H*2 *is a triple f* =*< f*0*, f*1*, f*2 *>, where f*0 : *I*2 → *I*1 *is an instance set morphism, f*1 =*< f*1*,*1*, f*1*,*2 *>*: S1 → S2 *is a transition system morphism, and f*2 *maps transitions of T*1 *to bMSC morphisms from λ*2 ◦ *f*1*,*2(*T* ) *to λ*1(*T* )*.*

Definition 4.4 (Fibered product of HMSCs) *Let H*0*, H*1 *and H*2 *be three HMSCs, and let f* =*< f*0*, f*1 =*< f*1*,*1*, f*1*,*2 *>, f*2 *>*: *H*1 → *H*0 *and g* =*< g*0*, g*1 =*< g*1*,*1*, g*1*,*2 *>, g*2 *>*: *H*2 → *H*0 *be two HMSC morphisms. The ﬁbered product of H*1 *and H*2 *over f and g is noted H*1 *f* ×*g H*2*, and is the HMSC H*1 *f* ×*g H*2 = (*I,* S*,* M*, λ*)*, where:*

* *I* = *I*1 *f*

+*g*0

*I*2*;*

0

* S = (*S, s*^*,* Σ*,T* ) *is the synch*}*ronous product of* S1 *and* S2 *under C* = (*t*1*, t*2)∈

*T*1 × *T*2 | *f*1*,*2(*t*1)= *g*1*,*2(*t*2)

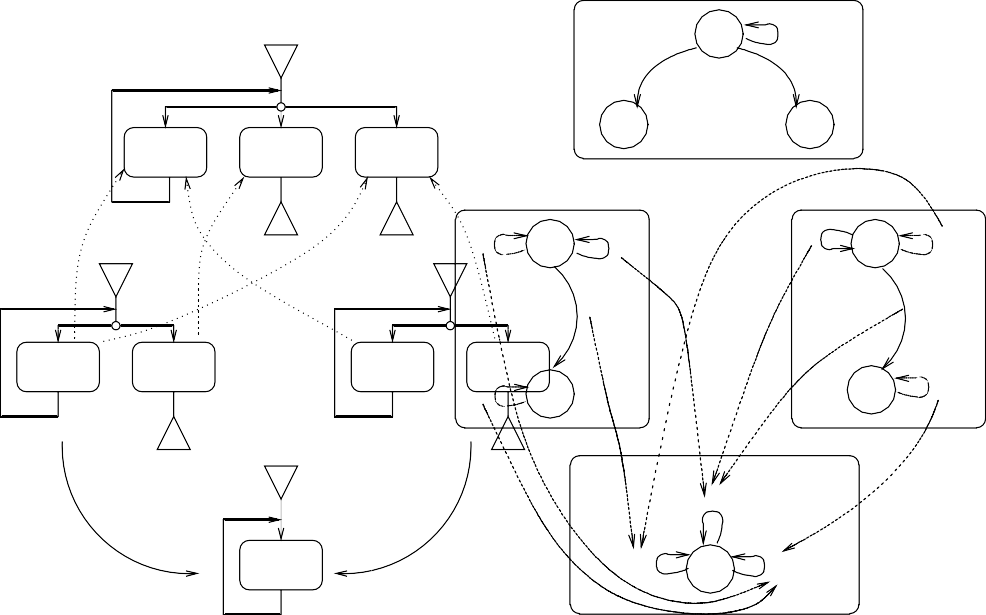
*;*

* *λ*(*t*1*, t*2)= *λ*1(*t*1) *f*2 (*t*1 ) +*g*2(*t*2 ) *λ*2(*t*2)*;*
* M = *λ*(*T* )*.*

Let us illustrate this notion of fibered product of HMSCs on an exam- ple. Figure [7](#_bookmark11) shows a product of two HMSCs *H*1 and *H*2 with two HMSC morphisms *f* =*< f*0*, f*1*, f*2 *>* that maps *H*1 to an interface HMSC *HI* and *g* =*< g*0*, g*1*, g*2 *>* that maps *H*2 to *HI*. The result of this product is given by HMSC *Result*. The real difficulty of HMSC product is the definition of an interface HMSC (*HI* in our example) and of the corresponding HMSC mor- phisms (*f* and *g*). As we just want to illustrate HMSC product, we will not detail the instance sets on which these HMSCs are defined nor the correspond- ing morphisms (*f*0 and *g*0). We will not either detail *f*2 and *g*2 that provide the bMSC morphisms used to build amalgamated sums *loop* + *Nominal* and *loop* + *deadlock*, and will only focus on the support automata product.

Let us detail the two LTS morphisms *f*1 and *g*1 that are used to build the product of *H*1 and *H*2’s support automata. The support automaton *SH*1 of *H*1 comports transitions with labels *l* and *t*, that are respectively mapped to bMSCs *Loop* and *terminate* by *λ*1 and the support automaton *SH*2 of *H*2 comports two transitions *n* and *d* that are respectively mapped to bMSCs *nominal* and *deadlock* by *λ*2. Following the convention on LTS, additional *ϵ*-transitions are added to the support automata. Morphism *f*1 sends respectively the transition labeled *l, t, ϵ*0*, ϵ*1 to those labeled *i, ϵ*0*, ϵ*1*, ϵ*1. In Figure [8](#_bookmark11), morphisms *f*1 and *g*1 are symbolized by dashed arrows from transitions of *SH*1 (respectively *SH*2) to transitions of *SI*. For this example, the synchronization constraint *C* built from *f*1 and *g*1 is defined by *C* =

S*Result*



*s s*'

0 0

HMSC Result

(t,*ε*)

(l,n)

(l,d)

*s s*'

1 0

*s s*'

0 1

loop

loop

+ terminate +

nominal deadlock

S*H*1

*ε*0 *s*0 l

S*H*2

n

*s*0

'

*ε*0

HMSC H1

HMSC H2

t

d

loop

terminate

nominal

deadlock

*ε*1

*s*

1

*s*'

1

*ε*1

S*I*

HMSC I

f

g

i

interact

*ε*0 *sI ε*1

Fig. 7. product of HMSCs H1 *f* ×*g* H2

Fig. 8. Support automata for HMSCs Fig [7](#_bookmark11)

⎧⎨ (*s , l,s* )*,* (*s*' *, n, s*' ) ; (*s , l,s* )*,* (*s*' *, d, s*' ) ; ⎫⎬

0

0

0

0

0

0

0

1

,

⎩ (*s , t,s* )*,* (*s*' *,ε , s*' ) ; (*s ,ε ,s* )*,* (*s*' *,ε , s*' ) ; (*s ,ε ,s* )*,* (*s*' *,ε , s*' ) ⎭

0

1

0

0

0

0

0

0

1

1

1

1

1

1

1

1

1

The synchronous product of support automata S*H*1 × S*H*2 with *C* is given by the support automaton S*Result* of Figure [8](#_bookmark11). For the sake of clarity, empty transitions (pair of *ϵ*-transitions) are not represented on the product automa- ton. To obtain HMSC Result, the bMSCs associated to transitions of S*Result* can then be amalgamated. For a given transition (*t*1*, t*2) of the support au- tomaton of the fibered product, the bMSC *λ*((*t*1*, t*2)) is the amalgamated sum of the bMSC associated to transition *t*1 and the bMSC associated to transi- tion *t*2. The two bMSCs morphisms needed for amalgamated sum are given by *f*2(*t*1) and *g*2(*t*2).

# Non-local choices suppression

Let us illustrate our HMSC product on a concrete application. As mentioned in Section [2](#_bookmark1), some HMSCs can contain non-local choices. During implementa- tion, this abstraction is error prone. To implement a system described by such a HMSC, an experimented programmer would certainly add some communi- cation messages to the HMSC, so that any non-local choice would be turned

into a local one. More generally, this kind of control can be implemented via well known distributed consensus protocols.

The approach proposed in this section is to integrate automatically a con- sensus protocol described with HMSC in a non-local description of a system. This integration is done using the fibered product between the non-local choice HMSC and another HMSC describing the consensus protocol. An interesting property of this HMSC transformation is that it can be used for an arbitrary number of instances participating to a non-local choice.

Let us consider the non-local HMSC *H*1 of figure [2](#_bookmark3). This description can be transformed into a local one by insertion of a control protocol that will tell instances *A* and *B* which branch they should follow. This protocol should have a single minimal event for each branch of the choice, and this minimal event must be performed by the same instance to ensure locality of the choice. A first solution is that an instance among all participating instances is designated as a master, and initiates a token ring protocol involving all participating instances in a certain order. This token ring is used to elect the branch to be performed. A drawback of this solution is that the symmetry of the HMSC is lost.

Another solution is to add a supervisor instance to our HMSC. The role of the supervisor instance is to ask all instances which branch they want to perform, to select an answer among the responses received, and to transmit this value to all instances. Several decision policies can be used. However, a first arrived, first served policy is assumed in this paper. Let us try to use this second solution to transform HMSC *H*1. So, the local HMSC we should obtain after transformation will look like the HMSC of Figure [9](#_bookmark13).

This transformation raises two important issues. The first one is the cre- ation of an election protocol for an arbitrary number of instances. Indeed, to make such an approach usable, one cannot ask a user to design *n* scenarios when *n* instances participate in a non-local choice. Fortunately, building such scenarios can be accomplished as an application of the amalgamated sum of bMSCs, and can be completely automated. The second issue is the insertion of the protocols right after choice nodes. This can be accomplished using the fibered product of HMSCs described in section [4](#_bookmark7). For this application, the interface HMSC is simple and can be computed automatically.

* 1. *Consensus Protocol construction*

As mentioned before, we have chosen to add a supervisor with a first arrived, first served policy to ensure locality of choices. The main idea is to insert a preamble *Ci* on each branch *Bi* of a non-local choice. Each preamble is a protocol in which the supervisor sends messages to all instances that are participating to the non-local choice. The first instance *i* that answers receives

HMSC H

M0

C1

C2

M1

M2

bMSC M0

A

B

m0

bMSC C1 bMSC C2

A

B

Supervisor

newchoice

newchoice

answer(1)

choice(1)

answer(x)

choice(1)

A

B

Supervisor

newchoice

newchoice

answer(x)

choice(2)

choice(2)

answer(2)

bMSC M1 bMSC M2

A

B

m2

Fig. 9. Local HMSC

A

B

m1

an acknowledgment of its choice, and all others receive a notification of the choice of instance *i*. The preamble protocol is depicted in Figure [10](#_bookmark14).

However, for a non-local choice involving up to *n* branches, a designer would have to define up to *n* quasi-identical preambles, which is a tedious work, and can be error-prone. Fortunately, these preambles can be constructed automatically using the amalgamated sum. First, let us show how a preamble can be constructed for two instances. The only bMSCs needed are: a generic bMSC *C Generic* with 3 instances (a supervisor, the first instance to answer, and the last instance) and an empty bMSC *M* over the participating instances. Figure [11](#_bookmark15) shows two sums with such kind of empty bMSCs. As the generic bMSC already contain 3 instances, the only task to perform to build our preamble is a renaming of *F irst* and *Z*. This can be done with the sum *M f* +*g C Generic*, where *f* : *M Interface* '→ *M* is the identity morphism,

bMSC Preamble

Fig. 10. Description of a preamble

First

*I*1

*in*−1 Supervisor

newchoice

newchoice

*answer*(*p*1 )

*answer*(*pn*−1 )

choice(n)

choice(n)

choice(n)

newchoice

answer(n)

and *g* = (*g*0*, g*1*, g*2): *M Interface* → *C Generic* is a morphism where *g*1 and *g*2 are empty functions and *g*0 is an instance set morphism that associates respectively *A* to *F irst* and *B* to *Z* on one side and *B* to *F irst* and *A* to *Z* on the other side. With no surprise, our sums produce bMSCs *C*1 and *C*2 of Figure [9](#_bookmark13), which are only renamings of the generic bMSC. As all our preambles will be isomorphic bMSCs, it is however important to note that renaming can be expressed using an amalgamated sum.

bMSC M

bMSC C\_Generic

bMSC M

Fig. 11. Preambles for a pair of instances

B

A

newchoice

newchoice

answer

choice(X)

bMSC M\_Interface

A B

bMSC M\_Interface

B A

choice(X)

answer

B

first

A

Z

Supervisor

* 1. *Generalization*

So far, it has been shown how to rename a generic preamble in the specification of a consensus protocol involving two instances and two different branches. This protocol can be generalized to an arbitrary number of instances. Let us note *C Generic*(*n*) a preamble involving *n* instances as depicted in Figure [10](#_bookmark14). First, let us show that there exists an interface MSC *C Interf ace* and two morphisms *f* : *C Interface* −→ *C Generic*(2) and *g* : *C Interf ace* −→

*C Generic*(2) such that *C Generic*(3) = *C Generic*(2) *f* +*g C Generic*(2). The two bMSCs *C Interface* = (*EI,* ≤*I, AI, αI, φI,* ≺*I*) over *II* and *C Gener*- *ic*(2) = (*E*2*,* ≤2*, A*2*, α*2*, φ*2*,* ≺2) over *I*2 are depicted in Figure [12](#_bookmark16). *f* : *C Inter*- *f ace* → *C Generic*(2) is defined by the triple *f* =*< f*0*, f*1*, f*2 *>*, where:

* + - *f*0 : *II* → *I*2 is the identity morphism;
    - *f*1 : *EI* → *E*2 respectively maps *eI*1*, eI*2*, eI*3*, eI*4*, eI*5*, eI*6 to *e*21*, e*22*, e*25*, e*26*, e*29*, e*210
    - *f*2 : *AI* → *A*2 is the identity morphism.

bMSC C Generic(3)

first

Z1

Z

Supervisor

newchoice

newchoice

newchoice

answer

answer

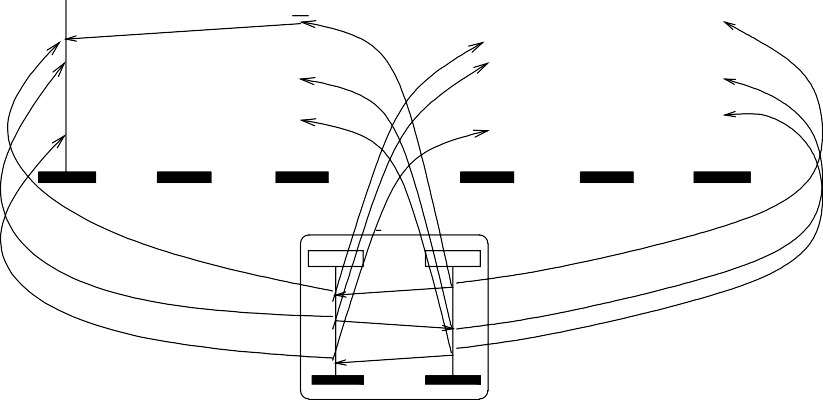
answer

choice(X)

choice(X)

choice(X)

bMSC C Generic(2) bMSC C Generic(2)



first

Z1

Supervisor

first

Z2

Supervisor

*e*22 newchoice

*e*2

answer

newchoice

*e*21

*e*23

*e*26

*e*2

*e*5

newchoice

answer

*e*

4

newchoice

*e*25

*e*1 *e*3

*e*6

*e*2

*e*2

*e*

7

*e*210

choice(X)

*e*212

*e*29

8

*e*

*e*9

8

*e*2

11

*e*10

choice(X)

*e*12

*e*

11

bMSC C Interface

first Supervisor

*eI*1

*eI*2 newchoice

answer

*eI*3

*eI*

4

*eI*6

choice(X) *eI*5

choice(X)

answer

choice(X)

answer

7

4

Fig. 12. the amalgamated sum *C Generic*(2) + *C Generic* = *C Generic*(3)

*f* and *g* are isomorphic morphisms, and *g* is defined similarly from *C Interface* to another copy of *C Generic*(2). Figure [12](#_bookmark16) illustrates the construction of C Generic(3) with bMSCs C Generic(2), C Interface, and bMSC morphisms *f* and *g*. This construction can be generalized to build a bMSC *C Generic*(*n*), that supervises the choices of *n* instances, using the bMSCs C Interface and C Generic of Figure [12](#_bookmark16), and two bMSC morphisms *fn*−1 and *gn*−1. It is de-

fined by induction: *C Generic*(*n*)= *C Generic*(*n*−1)*fn*−1 +*gn*−1 *C Generic*(2), where ∀*n* ∈ N, *gn* is the bMSC morphism *g* : *C Interface* → *C Generic* de- fined for two instances, and *fn*−1 = (*id, f*2*, id*): *Interface* −→ *C Generic*(*n* − 1) is a morphism such that *f*2 maps: events *eI*1 and *eI*2 to two events *x* and *y* associated to the emission and reception of message *newchoice* from

*Supervisor* to *F irst*; events *eI*3

and *eI*4

to two events *x*' and *y*' associated to

the emission and reception of message *answer*(*p*) from *F irst* to *Supervisor*;

and events *eI*5

and *eI*6

to two events *x*'' and *y*'' associated to the emission and

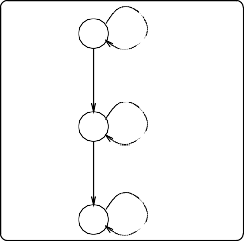
reception of message *Choice*(*X*) from *Supervisor* to *F irst*.

For a branch *Bi* such that *φ*(*min*(*Bi*)) = *j*, the preamble *Ci* will be a renaming of *C Generic*(*n*) through an instance renaming morphism that maps *j* to *f irst* and all other instances to any instance *Iq,q* ∈ 1*..n* − 1, and through a message renaming morphisms that modifies the message names.

* 1. *Insertion of preambles*

Once the desired preambles have been constructed and renamed, they have to be inserted in the original HMSC. This task is performed using the fibered product operation defined in section [4](#_bookmark7). For example, building HMSC *H* of Figure [9](#_bookmark13) from HMSC *H*1 amounts to the insertion of bMSCs *C*1 and *C*2 created in Figure[9](#_bookmark13) after the non-local choice node of *H*1. This insertion can be defined as a product *H*1 *f* ×*g H*2 between *H*1 and the HMSC *H*2 of Figure [14](#_bookmark18).

S1+2



*є*0 + *є*'

00

0

*є*0 + *b*

01

*є*0 + *є*'

1

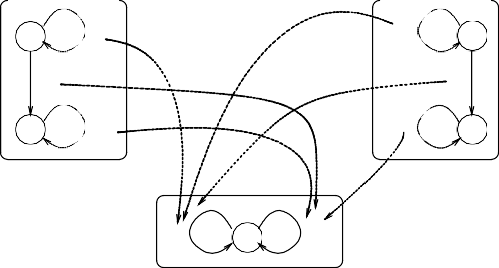
*a* + *є*0

11

*є*1 + *є*'

0

S1 S2



0

*є*0

*є*'

0

0

a

b

*є*1

*є*'

1

1

1

x

y

0

S*I*

Fig. 13. A simple insertion



HMSC H

M0

C1

C2

M1

M2

HMSC H1

HMSC H2

M0

C1

C2

M1

M2

g

f

HMSC H0

X0

X1

X2

X3

Fig. 14. Product H1 *f* ×*g* H2

The difficult part of insertion is the creation of an interface HMSC. Let us illustrate a simple insertion on an example. Consider the simple support automata S1 and S2 of Figure [13](#_bookmark17), and suppose we want to obtain a support automaton accepting sequence *b.a*. A way to “insert” transition *b* before transition *a* is to synchronize transition *b* with transition *ϵ*0 and transition *a* with transition *ϵ*' . This synchronization is implemented via the morphisms proposed in Figure [13](#_bookmark17).

1

The solution proposed for simple insertions can be generalized as shown in Figure [15](#_bookmark19). Let *f* be the HMSC morphism from *H*1 = (*IH*1*,* S*H*1*,* M*H*1*, λH*1) to *H*0 = (*IH*0*,* S*H*0*,* M*H*0*, λH*0) and *g* be the HMSC morphism from *H*2 = (*IH*2*,* S*H*2*,* M*H*2*, λH*2) to *H*0. More precisely:

* *f* : *H*1 → *H*0 is a triple *f* = *f*0*, f*1 =*< f*1*,*1*, f*1*,*2 *>, f*2 , where:
  + *f*0 : *IH*0 → *IH*1 is the identity; *f*1*,*1 maps all states of S*H*1 to *SI*;
  + *f*1*,*2 : *TH*1 −→ *TH*0 respectively maps (*s*0*, m*0*, s*1)*,* (*s*1*, ε*0*, s*1)*,* (*s*1*, m*1*, s*2) and (*s*1*, m*2*, s*3) to (*sI, x*0*, sI*)*,* (*sI, x*1*, sI*)*,* (*sI, x*2*, sI*) and (*sI, x*3*, sI*);
  + *f*2 : *T*1 → M0 → M1 associates a null bMSC morphism from *λ*0 ◦ *f*1(*t*) to

*λ*1(*t*) to each transition *t* ∈ *T*1.

* *g* : *H*2 → *H*0 is a triple *g* =*< g*0*, g*1*, g*2 *>*, where:
  + *g*0 : *IH*0 → *IH*1 is the identity; *g*1*,*1 maps all states of S*H*2 to *SI*;

S*H*

*s s*'

0 0

(*m , ε*' )

0 0

*s s*'

(*ε*0*, c*1) 1 0 (*ε*0*, c*2)

*s s*' *s s*'

1 1 1 2

(*m , ε*' ) (*m , ε*' )

1 1

*s s*'

2 2

*s s*'

2 1

S*H*1

*s*0

*ε*0 *m*0

*s*1

3 2

S*H*2 '

*ε*

*s*' 0

*ε*

*m*1 *m*2

*s*2 *s*3

*f*1 *x*1

' *c* 0 *c*

1

1

2

' '

*s*

*s*

1 2

S*H*0*x* '

*ε*

*g*

0

1 2

*sI*

) (

*x*2 *x*3

Fig. 15. labeled transition systems of HMSCs

* + *g*1*,*2 : *TH*2 −→ *TH*0 respectively maps (*s*' *, ε*' *, s*' )*,* (*s*' *,c , s*' )*,* (*s*' *,c , s*' )*,* (*s*' *, ε*' *, s*' )

0 0 0

0 1 1

0 2 2

1 1 1

and (*s*' *, ε*' *, s*' ) to (*sI,x ,s* )*,* (*s ,x ,s* )*,* (*s ,x ,s* )*,* (*s ,x ,s* ) and (*s ,x ,s* );

2 2 2

0 *I I* 1 *I*

*I* 1 *I*

*I* 2 *I*

*I* 3 *I*

* + *g*2 : *T*2 → M0 → M2 which associates a null bMSCs morphism from

*λ*0 ◦ *g*1(*t*) to *λ*2(*t*), to each transition *t* ∈ *T*2.

Note that a transition (*s, εs, s*) is artificially added to each state. These transitions allow independent moves of transition systems. Of course, *λ* will associate an empty bMSC *Mє*(bMSC with no instance or event) to each *ϵ*- transition. With this convention, we can define two bMSC morphisms *f* : *Mє* −→ *λ*(*t*) and *g* : *Mє* −→ *Mє*, such that *λ*(*t*) *f* +*g λ*(*ε*)= *λ*(*t*).

With morphisms *f*1 and *g*1 defined above, we can build the following syn- chronization constraint *C* = ((*s ,m ,s* )*,* (*s ,x ,s* )); ((*s ,ε ,s* )*,* (*s*' *,c , s*' ));

0

0

1

*I*

0

*I*

1

0

1

0

1

1

((*s ,ε ,s* )*,* (*s*' *,c , s*' )); ((*s ,m ,s* )*,* (*s*' *, ε*' *, s*' )); ((*s ,m ,s* )*,* (*s*' *, ε*' *, s*' )) . The

1

0

1

0

2

2

1

1

2

1

1

1

1

2

3

2

2

2

partially synchronized product S*H*1 × S*H*2 under *C* is the support automaton S*H*. Hence, with HMSC *H*0, *H*1 and *H*2, and with the definition of trivial HMSC morphisms *f* and *g*, we obtain the local HMSC *H* of Figure [9](#_bookmark13).

To summarize, from a non-local HMSC *H*, one can produce a HMSC

*Protocol* which bMSCs are built as a succession of amalgamated sums of

a generic bMSC, followed by a renaming. Then, the insertion of *Protocol* into *H* is performed with a product. All interfaces and morphisms used for this “localization” are simple and independent of the number of instances, hence this procedure can be completely automated for an arbitrary number of instances.

# Conclusion

This paper has proposed a formal notion of merge for bMSCs and HMSCs. This definition is based on the notions of synchronous product for transition systems and amalgamated sum. The use of this formalism on a concrete example shows the applicability of the method.

The definition of an HMSC product mainly relies on the definition of appro- priate morphisms, which can be considered rather complicated at first sight. Furthermore, morphisms force some synchronization between bMSCs, which can be considered as an arbitrary solution (remember that due to the meaning of sequential composition, there are several ways to obtain similar orderings between events, and hence that decomposition of scenarios into bMSCs is not always meaningful). A clue to refine merging is to define HMSC morphisms not only as transition morphisms but rather as path morphisms, as is done in [[3](#_bookmark23)]. Note also that amalgamated sums define explicitely events that co- incide in both views. This can be considered as a drawback, but relying on events labeling to detect similarities in views is not possible, as in the general case this approach exhibits several matching possibilities (and even an infinity when HMSCs are considered). However, we believe that for most cases where our product can be applied, the morphisms are rather simple, and can even be computed automatically. For the non local choice case, all morphisms are trivial, and do not use the full expressive power of synchronized HMSC prod- uct. Furthermore, the construction of local HMSCs from non local ones can be completely automatized.

An advantage of this composition method is that it allows the inductive definition of merging, and hence provides composition schemes for arbitrary number of instances. This product also defines a rough notion of coherence : if the HMSC obtained after composition is well formed, then the two operands are coherent on the set of events identified through the morphisms. If not, the fusion of both views are inconsistent.

Clues for future works are of practical and theoretical nature. On the practical side, we want to test our product through the definition of product- based transformation patterns for HMSCs. Of course, we do not plan to ask users to define their own fibered product of HMSCs, but rather to provide

libraries of formally defined patterns, than can be reused after instantiation of a limited number of parameters. Composition can also be used to create automatically concrete scenarios involving numerous instances that cannot be designed by hand from generic scenarios. These concrete scenarios could then be used for deployment, test and simulation purposes.

On the theoretical side, an interesting work consists in studying in detail the properties of algebraic operations defined as peculiar instances of product. This can be the base for a categorically well founded algebraic framework for scenario composition. Another interesting research direction is to study merging in a category of extended MSC (compositional MSCs [[7](#_bookmark27)] for example). The relation between HMSC morphisms and projections is also an inter- esting research topic. A question that seems quite natural is whether HMSC morphisms are inverse projections of HMSC, as defined in [[6](#_bookmark26)]. The answer is no, as morphisms do not preserve events labeling. When considering only label preserving morphisms, the answer is still no : interfaces are always HMSCs but HMSC projections are not HMSCs anymore as they can generate infinite connected patterns that can not be represented as compositions of a finite set of bMSCs. However, the relationship between projections and morphisms is something that is worth being studied, as finding a correct morphism is some- times equivalent to finding an appropriate projection. The ideal case is when

two HMSCs project on the same interface HMSC.

# References

1. Andr´e Arnold. *Finite transition systems*. Prentice-Hall. Prentice-Hall, 1994.
2. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Proceedings of TACAS’97*, LNCS no 1217, pp 259 – 274, Enschede,The Netherlands, April 1997. Springer-Verlag.
3. M.A. Bernarczyk, L. Bernardinello, B. Caillaud, W. Pawlowski, and L. Pomello. Modular system development with pullbacks. Technical Report 4828, INRIA, May 2003.
4. B. Caillaud and L. H´elou¨et. A categorical approach to hmsc composition. Research report, IRISA/INRIA, Nov. 2003. to appear.
5. A. Engels. Message refinement: Describing multi-level protocols in MSC. In Y. Lahav,

A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of SAM98:1st conference on SDL and MSC*, number 104 in Informatik-Berichte, pp. 67–74, Berlin, Germany, June 1998.

1. B. Genest, L. H´elou¨et, and A. Muscholl. High-level message sequence charts and projections. In *Proceedings of CONCUR’2003*, 2003.
2. E. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. In *Proc. of TACAS’01*, number 2031 in Lecture Notes in Computer Science, pp. 496–511, 2001.
3. D. Harel and R. Marelly. *Come, Let’s Play : Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
4. L. H´elou¨et and C. Jard. Conditions for synthesis of communicating automata from hmscs. In Axel Rennoch (Eds.) Stefania Gnesi, Ina Schieferdecker, editor, *Proc. of FMICS’2000*, Berlin, April 2000. GMD FOKUS. <http://www.gmd.de/publications/report/0091>.
5. L. H´elou¨et, C. Jard, and B. Caillaud. An effective equivalence for sets of scenarios represented by hmscs. Technical Report 3499, INRIA, September 1998. <ftp://ftp.inria.fr/INRIA/publication/RR/RR-3499.ps.gz>.
6. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1999.
7. J.P. Katoen and L. Lambert. Pomsets for message sequence charts. In *Proc. of SAM98:1st conference on SDL and MSC*, pp. 281–290, Berlin, Juin 1998.
8. Martin Leucker, P. Madhusudan, and Supratik Mukhopadhyay. Dynamic message sequence charts. In M. Agrawal and A. Seth, editors, *Proc. of FSTTCS’02*, LNCS no2556. Springer, December 2002.
9. S. Leue and P. Ladkin. Four issues concerning the semantics of message flow graphs. Technical report, INRIA Lorraine, 1994.
10. Sjouke Mauw and Michel A. Reniers. Refinement in interworkings. In *International Conference on Concurrency Theory*, pp. 671–686, 1996.
11. M. Reniers. *Message Sequence Charts: Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, 1998.
12. E. Rudolph, P. Graubmann, and J. Grabowski. Message Sequence Chart: composition techniques versus OO-techniques - ‘tema con variazioni’. In R. Bræk and A. Sarma, editors, *SDL’95 with MSC in CASE*, Proc. of 7th SDL Forum, pp 77–88, Oslo, 1995. Amsterdam, North-Holland.
13. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts.

*Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.