Formal Composition of Distributed Scenarios

Aziz Salah[1], Rabeb Mizouni[2], Rachida Dssouli[3], Benoît Parreaux[4]

[1]Department of C.S., University of Quebec at Montreal
Email: salah.aziz@uqam.ca
[2]Electrical & Computer Engineering, Concordia University
Email: mizouni@ece.concordia.ca
[3]Institute For Information System Engineering, Concordia University
Email: dssouli@ciise.concordia.ca
[4]France Telecom R&D, Lannion, France
Email:benoit.parreaux@rd.francetelecom.com

Abstract. *Eliciting, modeling, and analyzing the requirements are the main challenges to face up when you want to produce a formal specification for distributed systems. The distribution and the race conditions between events make it difficult to include all the possible scenario combinations and thus to get a complete specification. Most research about formal methods dealt with languages and neglected the process of how getting a formal specification. This paper describes a scenario-based process to synthesize a formal specification in the case of a distributed system. The requirements are represented by a set of use cases where each one is composed of a collection of distributed scenarios. The architectural assumptions about the communication between the objects of the distributed system imply some completions and reorganizations in the use cases. Then, the latter are composed into a global finite state machine (FSM) from which we derive a communicating FSM per object in the distributed system.*
Keywords*: Use case, Scenario-based approach, Scenario composition, Formal specification, Distributed systems, FSM.*

1. Introduction

The computer science community agrees that the requirement elicitation and analysis is a crucial step in the development process. Nevertheless, most research about formal methods dealt with languages and neglected the process of how getting a formal specification. Consequently, there is gap that makes difficult moving from requirements towards a formal specification. Developers avoid this phase by passing directly from informal requirements to implementation. Unspecified reception, service denial, and deadlock are common bugs that may have uncontrolled consequences in the case of distributed systems. Detecting such bugs during the validation stage becomes difficult, reducing thus the reliability of the system and increasing the development costs.

Scenario approaches have been emerged to fill the gap and facilitate the construction of a formal specification by promoting a "Divide and Conquer" strategy. A distributed scenario is a sequence of actions representing an execution trace describing a partial behavior, and providing a system level functionality. The actions represent concurrent interactions between different system objects. The different scenarios have to be composed in order to provide a formal specification of the system.

The requirements are widely represented by use cases where each one depicts a collection of scenarios. A scenario can be described by a message sequence chart (MSC), which emphasizes the interactions among objects. Our objective is to synthesize finite state machines (FSMs) from a set of use cases. Constructing communicating FSMs from MSCs is a very hard problem because MSCs may represent incomplete and inconsistent requirements. Combinatorial complexity makes it difficult to express MSCs for all of the possible scenario combination in the system behavior. Furthermore, systems may have many infinite traces, which cannot be easily captured by having only the MSC model. As a result, during the synthesis of FSMs, the analyst uses ad hoc methods based on his creativity and his expertness to fill the gaps.

MSCs and FSMs share some information, but emphasize two different views of the system behavior. First, MSCs represent the specification that the system should respect while FSMs are a model of the specification. Second, an MSC describes a story in which only some objects participate. Hence, it provides an inter-object view which makes it suitable for test and validation activities, but not for the implementation. In contrast, an FSM shows an intra-objects view where all the stories are about the same object [7] and may reflect their implementation.

Assuming that the system is composed of a set of objects, we aim at automating the synthesis of FSMs from use cases. We propose a two-phase method. The first phase consists of generating MSCs from use cases for completing them with missing scenarios. The intended communicating FSMs should allow infinite runs but use cases describe only finite traces about the behavior. Therefore, the second phase consists of enriching the use cases with some information that captures loops in the behavior and allows a system state characterization used for the automatic synthesis of a communicating FSM per object.

The paper is structured as follows: in Section 2, we give an overview of the notation we are using. Section 3 presents the formalization of use cases and scenarios using the tree presentation as well as the derivation of their MSCs. In Section 4 and 5 respectively, we describe the approach we are proposing for decorating use cases and synthesizing the communicating FSMs. Discussions on some related work are given in Section 6. Finally, Section 7 closes the paper with conclusions and future work.

2. Preliminaries, definitions and formal semantics

Let $\Omega = \{O_1, O_2, \ldots, O_n\}$ be the set of objects in the targeted distributed system, *Env* its environment, and $A_{O_i} = (S_{O_i}, S_{O_i init}, T_{O_i})$ the FSM of object $O_i \in \Omega$. S_{O_i} is the set of states of O_i, $S_{O_i init}$ is the set of initial states of O_i, and $T_{O_i} \subset S_{O_i} \times \Sigma_{O_i} \times S_{O_i}$ is the set of transitions where Σ_{O_i} is set of labels in the form $(O_i, O_j.m)$ or $(O_j, O_i.m)$. The FSM has a powerful capability of abstraction needed during first stages of the development process.

In this work, the FSMs of objects are assumed to be communicating FSMs according to the semantics of input/output automata defined in [12]. Each FSM object is autonomous and can communicate with other FSMs by means of message exchange. When an FSM object sends a message to another FSM, the latter is assumed to be ready to receive this message; otherwise there is an unspecified reception fault. The communication between FSMs is modeled by their parallel composition FSM denoted by $\prod A_{O_i} = (S, S_{init}, T)$. It is defined as the connected components of the composition FSM of $A_{O_1}, A_{O_2}, \ldots$ and A_{O_n}, which contains a state from S_{init}, where $S = S_1 \times S_2 \times \ldots \times S_n$, and $S_{init} = S_{O_1 init} \times S_{O_2 init} \times \ldots \times S_{O_n init}$. $T \subset S \times \Sigma \times S$ is the set of transitions of $\prod A_{O_i}$ defined by the following rules:

- Rule 1 : $(s_i, a, s_i') \in T_{O_i}$ *and* $(a=(O, O_i.m)$ *or* $a=(O_i, O.m))$ *and* $O \in \{O_i, Env\}$ ***implies*** $((s_1,..,s_{i-1},s_i,s_{i+1},..,s_n), a, (s_1,..,s_{i-1},s_i', s_{i+1},..,s_n)) \in T$

- Rule 2: *let* $O_i, O_j \in \Omega$ *and* $i<j$
$(s_i, a, s_i') \in T_{Oi}$ *and* $(s_j, a, s_j') \in T_{Oj}$ *and* $(a=(O_i, O_j.m)$ *or* $a=(O_i, O_j.m))$ ***implies*** $((s_1,..,s_i,..,s_j,..,s_n), a, (s_1,..,s_i',..,s_j',..,s_n)) \in T$

Rule 1 treats internal actions or communication with the environment while Rule2 treats communication among two different objects O_i and O_j.

Message sequence charts (MSCs) [9] are a commonly used visual representation of scenarios expressing the interactions among objects, components or processes. An MSC focuses on message exchange and shows a partial order of events. A message represents an interaction between two objects, a sender and a receiver. MSCs may display an order of events which is not always the only case supported by the implementation of MSCs. The formalization of MSCs allows the definition of the real partial order according to particular architectural communication assumptions. The formalization of MSC was traited by many reseachers [3, 4], and we propose a similar approach.

We formalize an MSC as a structure $(I, SE, RE, r, L, p, <_D, <_m)$ where
- $I \subset \Omega \cup \{Env\}$ *is a set of objects*

- *SE is the set of sending events and RE the set of receiving events. We denote by SE_O (respectively RE_O) the set of sending events (respectively the set of receiving events) in object O*
- *$r : SE \rightarrow RE$ maps a sending event to its receiving event. r is a bijection.*
- *L is a set of labels of the messages in the MSC*
- *$p : SE \cup RE \rightarrow I$ maps a sending event or receiving event to an object from I*
- *$<_D = \cup_{O \in I} <_O$ where $<_O \subset SE_O \cup RE_O \times SE_O \cup RE_O$ is a total order between the local events in object O according to the visual order as displayed in the MSC.*
- *$<_m = \{(s, r(s)) \mid s \in SE\}$ is an ordering relation which means that a message cannot be received before it is sent.*

The previous definition is very general and does not include any assumption about the communication architecture in the system. As the behavior described by MSCs will be translated into a set of communicating FSMs, their respective semantics should be compatible. Since the communication between FSMs, as defined in this paper, has no buffering facility, we assume that the FIFO order is preserved when an object receives two or more messages from the same object. Thus, the events should fulfill the partial ordering relation $<_{FIFO} = \{(r(s),r(s')) \mid (s,s') \in <_D$ *and* $p(s)=p(s')$ *and* $p(r(s))=p(r(s'))$ *and* $s \in SE$ *and* $s' \in SE\}$. Furthermore, FSMs are modeling autonomous objects. Consequently, an object has the control over its sending events. Hence, its scheduling of sending events is granted according to the visual order $<_D$. We also grant the local visual order between a receiving event and the next sending events in an object. Those facts are expressed by the following control ordering relation: $<_C = \{(e,s) \mid e \in SE \cup RE, s \in SE$ *and* $(e,s) \in <_O$ *and* $O \in I\}$. Finally, the interpretation of an MSC is given by the partial order relation $<$ defined as the transitive closure of the combination of the three partial ordering $<_C$, $<_m$ and $<_{FIFO}$:

$$< = (<_C \cup <_m \cup <_{FIFO})^*$$

3. Formalization of use cases and scenarios

A use case is used to describe a distributed functionality of the system as seen by actors (external users). The analyst usually builds use case diagrams, which emphasize the relationships between use cases. Then, she or he provides a textual description of the possible scenarios of each use case. This informal description is hard to be used in automatic processing of scenarios. Consequently, we conceived a formal model, which describes a use case by a tree of actions. The analyst constructs the tree of a use case by using either a depth-first or a breadth-first strategy in order to get a complete description according to the current requirements. As the use case tree paths are the scenarios of running the use case, the depth-first strategy is more convenient from a user point of view. However, the breath-first strategy is suitable to check that all of the possible scenarios have already been included in the use case tree since after each action all the possible afterward actions are checked. Actions (also called messages) are labels like $(O_i,O_j.m)$ where O_i and O_j are objects of the system. $(O_i,O_j.m)$ means that message m is sent from O_i to O_j.

We will be using a basic telephone system to illustrate our work. Fig.1 shows use case "Make a call" that describes the behavior of the system when a *user A* calls a *user B*. We will assume that Ω for the telephone system is composed of the following objects: *A*, *B* and a switch S. Let's now present the formal definition of our use case model:

Definition: a use case Γ is a tree $\Gamma=<Id,M,M_{start},Parent>$ where:
- $\Gamma.Id$ is the id of the use case,
- $\Gamma.M \subset (\Omega \cup \{Env\}) \times (\Omega \cup \{Env\}.Label)$ *is the set of messages in the form (O,O'.m),*
- $\Gamma.M_{start} \subset \Gamma.M$ *is the set of starting messages,*
- $\Gamma.Parent$ *is a function that associates to a message the index of its parent message. Function* $\Gamma.Parent$ *is not defined for starting messages.*

The scenarios of a use case are complete paths starting from a *start message* and ending at one of the tree leaves. From each use case scenario, an MSC is generated. The generation of MSCs is only based on the syntax of messages and their order in the use case tree paths. The syntax of message label identifies the sender and the receiver objects. The use case tree of Fig.1 contains three scenarios. We have drawn in Fig.2 the MSCs generated from use case "Make a call".

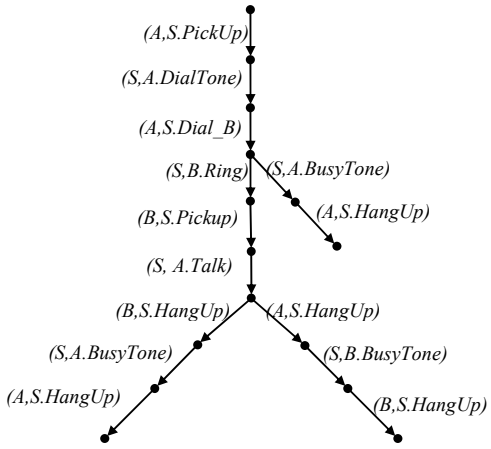

Fig. 1. The tree of use case "Make a call"

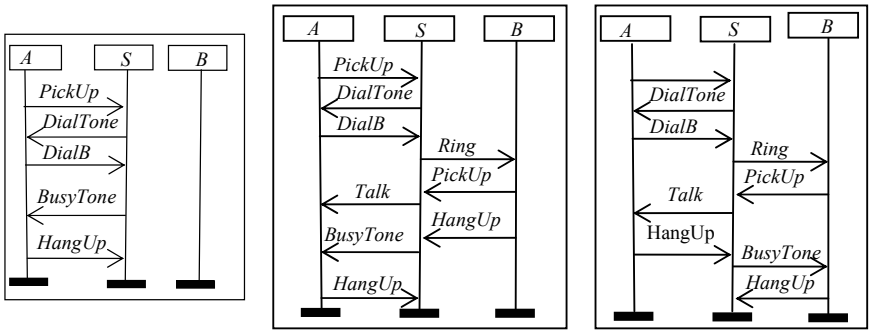

Fig. 2. The MSCs generated from the use case in **Fig. 1.**

MSCs are less intuitive than expected. Their visual order does not always represent all their possible executions as only a partial order of events is garanted according to a particular adopted semantics of MSCs. For this reason, researchers defined the notion of MSC linearizations [2] [15] to represent the possible executions of an MSC.

In this work, the linearization of an MSC is a total order relation which is consistent with its partial order relation <. If an MSC has many linearizations, some of them may not be included in the use case tree since they may have escaped to the user requirements. Thus, the partial order of an MSC helps the analyst to detect and possibly complete the use case tree by adding those absent linearizations after the user validation as illustrated in Fig. 3. If the user refuted one of the linearizations of an MSC, it means that the use case tree should be modified so that its generated MSCs accept no more the refuted linearization.

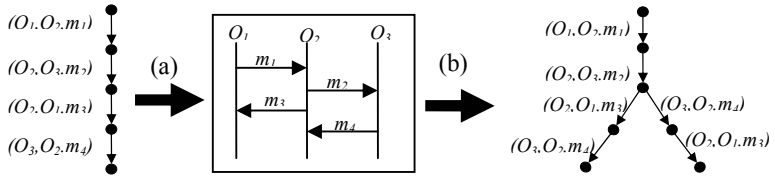

Fig. 3. Use case completion by adding absent linearizations
step (a): the MSC generation
step (b): adding a missing linearization to the use case

The user may sometimes be confused and does not realize that his use case is composed of a combination of independent traces. To formally define what independent traces means, let first define the set of minimum event *Min* of an MSC:

$$Min=\{e \in SE \mid \forall e' \in SE \cup RE \,.\, (e',e) \notin <\}$$

Min denotes the set of sending events where each one may initiate a sequence of events, called independent trace. If *Min* is not a singleton, the partial order of an MSC may be used to find out independent traces.

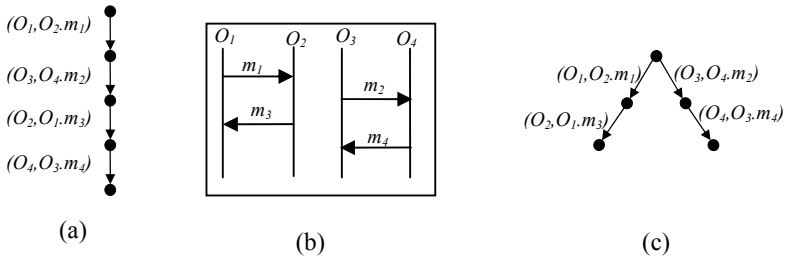

Fig. 4. Use case reorganization process: (a) Original use case tree. (b) Its generated MSC. (c) The proposed reorganized use case tree

If the generated MSCs of a use case include many independent traces as shown in Fig.4 (b), the computation of the set *Min* allows a reorganization of the use case so that the causality relationship between the independent traces becomes explicit as illustrated in Fig.4 (c). However, if the user refuted the proposed reorganization of the use case, it may mean that there are parts of the use case that are missing and should establish the causality relationship he intended.

4. Decorating use cases with a state characterization

The compatibility of use cases and their generated MSCs is reached when both of them accept the same scenarios. Our goal is to synthesize FSMs from use cases. As known, it is possible to generate FSMs from a set of traces. However, the behavior provided by use cases is partial since they don't include infinite traces or repetitive behaviors. High-level MSCs (HMSCs) are MSC-graphs where each node is an MSC [9]. They provide a mean to define how MSCs can be combined and they can express infinite traces of the system behavior. However, HMSCs specify an explicit combination of MSCs, available only in an advanced stage during system requirement analysis. Therefore, we adopted an alternative strategy that consists of decorating use cases with a state characterization. The latter allows not only capturing infinite traces, but also recognizing shared states in different scenarios and thus determining their relationships as well as the relationship between their respective use cases.

Decorating use cases gives the analysts the opportunity to add their interpretations regarding the state of the system when an action is performed. It consists of specifying for each message (action) of the use case partial pre and partial post conditions expressed by state variable constraints. Those conditions are qualified to be partial because they have to be completed by the fact that this action takes place before and after specific actions in the use case. State variables are defined by the analyst and their values represent the state of the system. As the latter is composed of a set of objects, the state of the system is also composed of the states of its objects. Subsequently, the state of an object can be derived from the global system state.

In practice, state variables have symbolic names. However, we will use here a vector-based notation because it is more convenient to present the general case. The state of the system is represented by a state vector $V=(v_1,v_2,..,v_k)$ where v_i is the *value* of state variable $V[i]$ and k is the number of state variables. We write $dom(V[i])$ the finite domain of possible values of state variable $V[i]$. A state variable may also be instantiated with a special value, denoted by *nil,* which means that its current value is not fixed yet in that state. Hence, the space of state vectors is the product set $DOM=(dom(V[1])\cup\{nil\})\times(dom(V[2])\cup\{nil\})\times\ldots\times(dom(V[k])\cup\{nil\})$.

The decoration of a use case consists of specifying for each message three declarative attributes: a partial pre-condition, a partial post-condition, and an extension point. The

Index(msg)	*Parent(msg)*	*msg*	*ppre(msg)*	*ppost(msg)*
0	-	*(A,S.PickUp)*	*SigA=N and StaA=I and SigB=nil and StaB=nil*	*StaA'=B*
1	*0*	*(S,A.DialTone)*	*True*	*SigA'=DT*
2	*1*	*(A,S.DialB)*	*True*	*SigA'=D*
3	*2*	*(S,A.BusyTone)*	*StaB=B*	*SigA'=BT*
4	*3*	*(A,S.HangUp)*	*True*	*SigA'=N and StaA'=I and SigB'=nil and StaB'=nil*
5	*2*	*(S,B.Ring)*	*SigB=N and StaB=I*	*SigB'=R and StaB'=B*
6	*5*	*(B,S.PickUp)*	*True*	*SigB'=T*
7	*6*	*(S,A.Talk)*	*True*	*SigA'=T*
8	*7*	*(B,S.HangUp)*	*True*	*SigB'=nil and StaB'=nil*
9	*8*	*(S,A.BusyTone)*	*True*	*SigA'=BT*
10	*9*	*(A,S.HangUp)*	*True*	*SigA'=N and StaA'=I*
11	*7*	*(A,S.HangUp)*	*True*	*SigA'=N and StaA'=I*
12	*11*	*(S,B.BusyTone)*	*True*	*SigB'=BT*
13	*12*	*(B,S.HangUp)*	*True*	*SigB'=nil and StaB'=nil*

Table 1. Decoration of use case "*Make a call*". The state vector is composed of the values of four variables *SigA* , *StaA*, *SigB* and *StaB*. *SigA* describes signals of *terminal A, and Dom(SigA)={N,DT,D,BT,T} where N* means *no signal, DT dial tone signal*, *D* dialing *signal*, *BT busy tone signal* and *T talking signal*. *StaA* describes the status of *terminal A and Dom(StaA)={I,B}* where *B* stands for *busy* and *I* for *idle*. Variables *SigB* and *StaB* describe respectively the signals and the status of *terminal B*. *Dom(SigB)={N,BT,R,T}* where R stands for ring signal and the other values are the same like in *Dom(SigA). Dom(StaB)={B,I}.* We have also decided that *EP* is set to *False* for all messages in this use case tree.

partial pre and post-conditions of a message *m* are denoted by *ppre(m)* and *ppost(m)* respectively. The state variables must fulfill the constraints *ppre(m)* before sending message *m* and *ppost(m)* after its reception. *ppre(m)* is a conjunction of elementary constraints in the form (*V[i]=v)* where *v* is a constant of *dom(V[i])*. In contrast, *ppost(m)* constraints the relation between the vector state *V* before *m* and *V'* the state after *m*. Thus, *ppost(m)* is a conjunction where elementary constraints are either *(V'[i]=v)*, *V'[i]= V[i] op v),* or *(V'[i]= V[I]),* and where *op* is an operator defined on the variables domain.

By default, a non-instantiated variable will be initially set to *nil*. Afterwards, we adopt the STRIPS [5] strategy to deal with the frame problem and assume all that is not explicitly changed by an action remains unchanged. Furthermore, whenever we have a conjunction[1] in the form *(V[i]=v and V[i]=nil)* the latter is unified to *(V[i]=v)*. This unification is needed later on in the computation of the canonical form of the use case.

[1] This conjunction differs from the ordinary logical AND since it provides a rewriting rule when a formula contains "*V[i]=nil"*.

Finally, the third element of the decoration is the extension point. Since the analyst has to compose many use cases to construct the overall system behavior model, we associate to each message *m* a predicate denoted by *EP(m) w*hich stands for extension point, similar to use case *extension point* in UML [16]. *EP* provides to the analyst a mean by which she or he can control how parts of FSMs coming from different use cases can be connected. By default, for a message *m* that it is not a leaf, the value of *EP(m)* is "*False"* in order to prevent overlapping use case traces. In contrast, the analyst decides which value should be assigned to predicate *EP* for other messages. If the *EP* is "*True*", it means that the execution of the system continues in the current use case. Otherwise, it indicates that the system may exit the current use case and continues its execution in another one. In this case, it represents the concatenation of use case traces. The decoration use case "Make a call" is presented in **Table 1**.

5. Synthesis of Communicating FSMs

Synthesizing communicating FSMs from decorated use case trees takes three steps: (1) transforming use cases into a canonical form, (2) synthesizing a global finite state machine (GFSM) from the canonical form of all use cases, (3) deriving from the GFSM a communicating FSM for each object in the system.

Step (1): Canonical representation of use case trees

The requirements of a system are composed of a number of use cases. Their overall behavior can be implemented by synthesizing communicating FSMs. For this end, we need a representation of use cases that not only captures their behavior, but also facilitates their merge into a global state model. We adopt thus a canonical representation of use cases in the form of a flat set of m-rules. An m-rule is an atomic message rule, which describes the states of the system before a message is sent and after it is received. Formally, an m-rule is a 3-tuple *mr=(LHS,RHS,lab)* where *mr.LHS* is the left hand side of the rule and represents the pre-condition part, *mr.RHS* is the right hand side which is the post-condition part, and *mr.lab* is the message synchronization label of the m-rule. We recall that *mr.lab* is in the form *(Oi,Oj.m)*.

In order to tag the states and the transitions in the targeted FSMs with the use case id from which they come, we extend the set of state variables with a new variable called *uc*. Tagging the FSMs is not only used for traceability reasons, but also to implement information given by predicate *EP* related to the extension points of a use case. From now and on, the state vector is composed of all state variable values and the value of the recently introduced variable *uc*. The domain of variable *uc* is the set of use case ids plus a special value denoted by *noUc* that tags the state vectors that may be shared by a certain number of use cases.

We define the canonical representation of a use case as a pair of m-rule sets denoted by $<R,R_{start}>$ and derived from the use case. The algorithm at Fig.5 describes how $<R,R_{start}>$ is computed from a use case. As shown in lines (8) to (11) in this

```
Input    <Γ,pre,post,EP> where Γ=<Id,M,M_init,Parent>  is
a decorated tree with ppre, ppost, and EP
Output    <R,R_init>
(1) R:=∅; R_init:=∅
(2) For each msg ∈ Γ.M do
(3)         mr.lab:=msg
(4)         If msg∈ Γ.M_init then
(5)          mr.LHS:= ( ppre(msg) and  uc= noUc)
(6)         Else mr.LHS:=(ppre(msg)and ppost(parent(msg)
                        and uc=Γ.Id )
(7)         For each msg'∈ Γ.M | msg=parent(msg') do
(8)          If EP(msg)=False then
(9)            mr.RHS:= ( ppost(msg) and ppre(msg')
                     and  uc=Γ.Id )
(10)         Else mr.RHS:= (ppost(msg) and ppre(msg')
                        and uc=noUc)
(11)         R:=R∪{mr} /*mr is added to R unless mr∉R*/
(12)         If msg ∈ Γ.M_init then R_init:=R_init∪{mr}
(13)        Done
(14) Done
```

Fig. 5. Computing the canonical form of a use case

algorithm, the same message may be duplicated into several m-rules such that each one would have an *RHS* that conforms the pre-condition in one of its next messages in the use case. Each extracted m-rule is tagged with the use case id by constraining its *LHS* and *LRS* with either the constraint *(uc=Γ.Id)* or the constraint *(uc=noUc)* according to the value of the predicate *EP* in its message. Hence, state vectors satisfying *(uc=Γ.Id)* are specific to use case *Γ*. In contrast, state vectors where we have *(uc=noUc)* shared by the use cases where the other state vector components coincide. Consequently, use cases having such state vectors may have their respective FSMs connected to each other by those shared state vectors. A conflict is reported to the designer whenever there is any m-rule from a use case that has false in either its *LHS* or *RHS* constraints.

Step (2): Synthesizing a global finite state machine from decorated use case trees

The global finite state machine (GFSM) is an FSM constructed from all use cases. Assuming that we have a communicating FSM for each object, the GFSM should represent the FSM of their parallel composition. The GFSM accepts at least all the complete paths of the use case trees.

In practice, we directly derive the GFSM of a use case from its canonical representation. Let $<S,S_{inil},T>$ be the GFSM of a use case for which the canonical

representation is $<R,R_{init}>$. Let *[r.LHS]* be the set of state vectors which verify the constraint *r.LHS* and *[r.RHS]* the set of pairs of state vectors that verify the constraint *r.RHS*. We define the GFSM $<S,S_{init},T>$ by the following:

$$T = \{(V, l, V') \mid \exists r \in R \,.\, V \in [r.LHS] \quad and\ (V, V') \in [r.RHS]\ and\ l = r.lab\}$$

$$S = \{V \in DOM \mid (V, \_, \_) \in T \quad or \quad (\_, \_, V) \in T\}$$

$$S_{init} = \bigcup_{r \in R_{start}} [r.LHS]$$

S is the set of states of the GFSM and composed of state vectors, which satisfy either the *LHS* or the *RHS* of an m-rule. *T* is the set of transitions. Each one comes from an m-rule. We point out that the GFSM can be non deterministic. The GFSM of use case "Make a call" is drawn in Fig.6, and its state vectors are described in **Table 2**.

We have so far treated the construction of the GFSM of one use case. The generalization to the case of two or more use cases consists of synthesizing the GFSM derived from the union of those use cases canonical representation. We define the union of two canonical representations $<R,R_{init}>$ and $<R',R_{init}'>$ as $<R \cup R', R_{init} \cup R_{init}'>$.

Step (3): Deriving communicating FSM for each object

The derivation of an FSM for an object consists of clustering some states and removing some transitions from the deterministic FSM (DGFSM) which correspond to the GSFM of all use cases. The DGFSM can be obtained from the GFSM by using

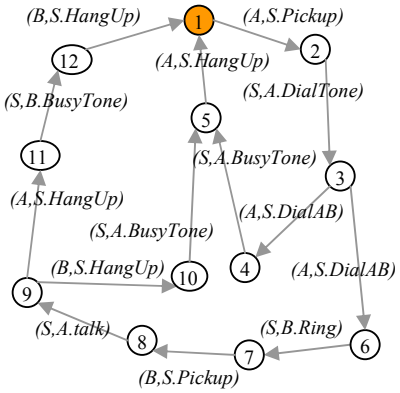

Fig. 6. GFSM of use case "Make a call"

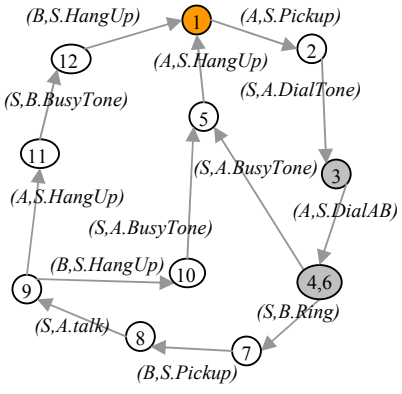

Fig. 7. DGFSM of use case "Make a call"

	1	2	3	4	5	6	7	8	9	10	11	12
SigA	*N*	*N*	*DT*	*D*	*BT*	*D*	*D*	*D*	*T*	*T*	*N*	*N*
StaA	*I*	*B*	*B*	*B*	*B*	*B*	*B*	*B*	*B*	*B*	*I*	*I*
SigB	*nil*	*nil*	*nil*	*nil*	*nil*	*N*	*R*	*T*	*T*	*nil*	*T*	*BT*
StatB	*nil*	*nil*	*nil*	*B*	*B*	*I*	*B*	*B*	*B*	*nil*	*B*	*B*
uc	*noUc*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

Table 2. State vectors of GFSM of use case "Make a call"

```
Inputs:
       - DGFSM <S,S_init,T>, { S_init is a singleton }
       - O an object in Ω
Output:
       -<S_0,S_0init, T_0>, the FSM of object O

T_0:=∅; S_0:=∅; S_0init:=∅
/*Clustering states */
For each (V,(O_i,O_j.m),V') in T do
       If (O_i≠O and O_j≠O) then
          S_0:=Cluster(V,V',S_0)
       Else
          S_0:=Cluster(V,V,S_0)
          S_0:=Cluster(V',V',S_0)
       fi
done
T_0:={(C,msg,C') | (V,msg,V') ∈ T and V∈C
                    and V'∈C'}
S_0init:= {C | C∈S_0 and  ∃ V∈S_init . V∈C}
Return(S_0,S_0init,T_0)

Where  Cluster(V,V',CS):
       If (∃ CE ∈ CS such that V∈ CE ) then C:=CE
       Else C:= ∅
       If (∃ CE' ∈ CS' such that V'∈ CE' ) then C':=CE'
       Else C:= ∅
       Return(CS\{C,C'}∪{C∪C'∪{V,V'}})
```

Fig. 8. Construction of the FSM of an Object

the algorithm given in [1]. We assume that the state of an object is supposed unchanged if no action occurs in that object according to the GFSM. Consequently, the FSM states of an object *O* are obtained by clustering into the same state all the states of the GFSM that are connected with a transition in which object *O* does not participate. The transitions of the FSM of an object *O* are only those transitions of the GFSM representing messages or actions in which the object *O* participates. This algorithm is presented in Fig.8. The FSM of an object implements all the parts of use cases in which that object participates. Consequently, the FSM of an object implements the object behavior.

We have constructed from the DGFSM (c.f. Fig.7) the FSMs of objects "*Terminal A*", "*Terminal B*", and "*Switch*". The FSM of object "*Switch S*" resulting from that algorithm is exactly the entire FSM in Fig.7. However, The FSMs of objects "*Terminal A*" and "*Terminal B*" respectively are drawn in Fig.9.

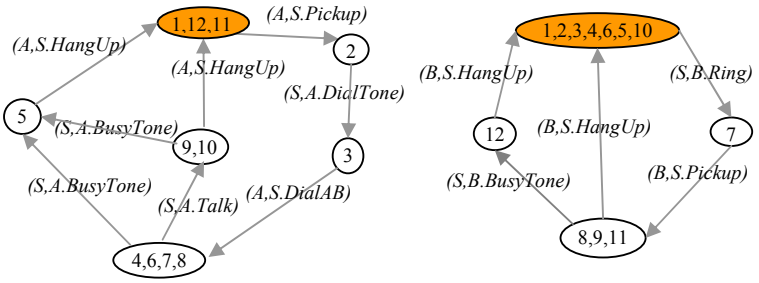

Fig. 9. The FSM of Terminal A (left side) and the FSM of the Terminal B (right side)

The FSM of an object represents its behavior as described by the input decorated use cases and it is not error free. The FSMs can be inspected for some patterns and may reflect some errors. For example, the states in the FSM of an object should not have any self-loop transition. The latter shows that the object accomplishes an action, but its state does not change, a contradictory fact to use case decoration assumptions. With the use case id in the state vector we can trace back exactly where the analyst should intervene to correct the anomaly.

6. Related work and discussion

Researchers have intensively investigated the transformation of scenarios into transition-based system model during the last ten years. To deal with these topics, the key idea is how to identify states at the scenario level such that those states can be recognized in different scenarios and then integrated in the target global model. There are two kinds of state characterization: trace-based [8, 10, 11, 13], and variable (or label) state-based characterization [17 , 18, 19]. In this paper we adopted the second approach to identify the states of the system.

Harel et al. [8] tackled the problem of synthesizing statecharts from LSCs (Live sequence charts), an extended form of MSCs which support liveness by specifying universal and existential scenarios. Their approach consists of synthesizing global automaton with accepting states from LSCs using trace-based state characterization. The global automaton can be decomposed into an automaton per object. The latter constitutes the overall statechart. Since we focus on first stages of requirement analysis, we believe that MSCs are easier to use with decoration and to validate. Moreover, practicing decoration is not compatible with LSCs because it may threaten their notion of universal scenarios.

The closest approaches, in terms of state characterization, to our are [17, 18, 20]. In the first one, Whittle et al. [20] captured domain information by specifying for each message type a pre and a post-condition once for all. Contrarily to what we propose, sequence diagrams SDs (a variant of MSCs) are transformed into an FSM per object

and per SD using a state-variable unification and propagation procedure. However, the state unification definition does not consider the causality relation between the unified states. The FSMs of an object are then merged into a single FSM based on defined state similarities. The authors introduced hierarchy into FSMs based on state variable ordering, class diagrams and generalization of transitions. Moreover, message passing is assumed to be hand shacking. Thus, the SDs would have only a single linearization.

The work presented in [17, 18] shows techniques for synthesizing timed automata from scenarios with respect to time constraints. Timed automata allow the description of the behavior of real time reactive systems. The scenarios can be seen as an enriched form of MSCs. The state characterization is very similar to the one we have presented in this paper. In contrast, the state vectors as defined in this paper are global, so they capture simultaneously the states of all objects. Giese [6] presented too an approach towards the synthesis of parametric timed automata from scenarios. Un-timed scenario are first derived according to existing approach like the one of Uchitel et al. [19], then the timing constraints are added in an incremental manner as time boundaries. The approach detects all the timing conflicts that can occur when integrating different scenarios, and hence can be adjusted. Contrarily to the approach in [18], Giese is synthesizing more than one automaton in the same time.

In most cases, the composition of scenarios ends up with the creation of unexpected implied scenarios. The latter could stress incompleteness in the specification so it is useful to add them to the use case, or they express undesired behaviors that have to be removed from the targeted model. Many researchers have tackled the problem of detection and elimination of implied scenarios [2, 13, 14, 19]. Alur et al. [2] have developed an algorithm to transform a set of MSCs into communicating state machines. Their approach has the potential of detecting all the implied scenarios from a set of MSCs. However, they consider only the case of finite traces. Uchitel *et al* [19] added the HMSCs to the specification so that they introduce the infinite execution aspects. Their approach consists of first constructing the labeled transition system of each object after what they compose them to obtain the overall implementation of the system. Our approach, however, uses the concept of decoration to detect loops and hence introducing the infinite behavior aspect in the specification.

An important issue is whether or not all implied scenario have to be eliminated. Some researchers make the choice to produce a specification that is the closest possible to the original use cases. Thus, they conduct their approaches in such a way they detect implied scenarios and eliminate them [14, 19]. Such decision has the advantage to be automatic. However, others [13] make the choice to return back to the user to accept or refute a detected implied behavior. The advantage here is the enhancement done to the original use case. We opted for the second alternative because we believe it has the potential to complete the system behavior.
Our approach differs substantially from the earlier presented work by the following points. We introduced an intermediate level of granularity, which is the level of use cases. Use cases themselves include a finer level of granularity represented by scenarios.

Furthermore, from the described behavior in the user case and using its generated MSCs, we detect the other unexpected scenarios due to race conditions in a distributed system. Those implied scenarios are detected by enumerating all the linearizations of the MSCs the use case does not accept. This procedure offers the opportunity to remove in an early stage the undesired behaviors and allows the user to complete his current use case. In addition, the decoration of a use case by state characterization is easier than the decoration of an isolated MSC because a use case provides a broader view that shows the relationships between its scenarios. Moreover, a message which belongs to several scenarios will be decorated only once in the use case.

Our approach distinguishes between two classes of implied scenarios: the intra-use case implied scenarios and the inter-use case implied scenarios. We define the former as a trace that the GFSM of a use case accepts but not its tree. This trace is the direct result of the use case decoration for which the role is to make possible such traces. The use case decoration configures the set of accepted traces to fit the user expectations. An inter-use-case implied scenario could be defined as a trace, which cannot be completed to correspond to a concatenation of complete path from different use cases. By construction, we can claim that there are no such implied scenarios in the GFSM of the system because of tagging private states in the use cases with their respective ids.

7. Conclusion

We have so far presented a method for constructing a communicating FSM from use cases expressed in the form of trees. MSCs are generated from use case trees and validated by users. The validation process consists of inviting the user to decide about accepting or not each one of the MSC linearizations missing in the use cases. Moreover, the user can also be prompted how to reorganize a use case in order to move forward a more structured specification. At the end of this stage, the original use cases may be modified to reflect more a desired and realizable system behavior. Use cases are then decorated for detecting repetitive behaviors and constructing their GFSM. Afterward, the latter is decomposed to derive a communicating FSM for each object.
The decoration of the use case trees seems difficult at first time, but with practice, analysts will develop skills to perform appropriate declarative decoration. Moreover, practicing decoration is very helpful for a well understanding of the requirements, especially in the case of distributed systems. Besides, even not explicitly shown, our approach preserves the traceability between use cases and the FSMs of objects. Hence, for any element (either a state or a transition) in the FSMs, we can retrieve the use case it is related to. So, when errors are detected in the FSM level, the analyst will be able to trace them back and correct them in the use case level. Our approach would be more efficient when implemented as a computer aided design tool with a graphical interface, which is under development.

Ackowledgments

This work has been supported France Telecom R&D through a contract between France Telecom and Concordia University. Aziz Salah is also supported by an NSERC grant.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Reading, Mass.: Addison Wesley, 1986.

[2] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," presented at 22nd International Conference on Software Engineering, 2000.

[3] R. Alur, G. Holzmann, and D. Peled, "An Analyzer for Message Sequence Charts," *Software: Concepts and Tools*, vol. 17, pp. 70-77, 1996.

[4] H. Ben-Abdallah and S. Leue, "Syntactic Detection of Process Divergence and Non-local Choice inMessage Sequence Charts," in *TACAS*, 1997, pp. 259-274.

[5] R. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artif. Intell.*, vol. 2, pp. 189-208, 1971.

[6] H. Giese, "Towards Scenario-Based Synthesis for Parametric Timed Automata," presented at the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE, Portland, USA, 2003.

[7] D. Harel, "From Play-In Scenarios To Code: An Achievable Dream," *IEEE Computer*, vol. 34, pp. 53-60, 2001.

[8] D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Int. J. of Foundations of Computer Science*, vol. 13, pp. 5-51, 2002.

[9] ITU, *Recommendation Z.120: Message Sequence Chart (MSC)*, 1999,.

[10] K. Koskimies and E. Mäkinen, "Automatic Synthesis of State Machines from Trace Diagrams," *Software-Practice and Experience*, vol. 24, pp. 643-658, 1994.

[11] I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," presented at Distributed and Parallel Embedded Systems, 1998.

[12] N. Lynch, "I/O Automata: A model for discrete event systems," presented at 22nd Annual Conference on Information Sciences and Systems, Princeton University, Princeton, N.J., 1988.

[13] E. Mäkinen and T. Systä, "MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML," presented at ICSE 2001, Toronto, Canada, 2001.

[14] H. Muccini, "Detecting Implied Scenarios analyzing non-local Branching Choices," presented at Conf. on Fundamental Approaches to Software Engineering (FASE 2003), ETAPS2003, Warsaw, Poland, 2003.

[15] M. Mukund, K. N. Kumar, and M. A. Sohoni, "Synthesizing distributed finite-state systems from MSCs," presented at Proc. CONCUR '00, 2000.

[16] OMG, "Unified Modeling Language (UML) specification v1.5," OMG document ad/2003-03-01, March 2003.

[17] A. Salah, R. Dssouli, and G. Lapalme, "Compiling real-time scenarios into a Timed Automaton," presented at FORTE XIV/PSTV XXI, 2001.

[18] S. Somé, R. Dssouli, and J. Vaucher, "Toward an Automation of Requirements Engineering using Scenarios," *Journal of Computing and Information*, vol. 2, pp. 1110-1132, 1996.

[19] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29, pp. 99-115, 2003.

[20] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," presented at 22nd International Conference on Software, 2000.