Branching Time Semantics for UML 2.0 Sequence Diagrams

Youcef Hammal

LSI, Département d'Informatique, Faculté d'Electronique & Informatique
Université des Sciences et de la Technologie Houari Boumediene
BP 32, El-Alia 16111, Bab-Ezzouar, Algiers, Algeria
yhammal@wissal.dz

Abstract. This paper presents formal definitions for UML Sequences Diagrams based on branching time semantics and partial orders in a denotational style. The obtained graphs are close to lattices and specify faithfully the intended behaviors rather than trace based semantics. We also define few generalized algebraic operations on graphs so that it makes it easy to provide formal definitions in a compositional manner to interaction operators. Next we extend our formalism with logical clocks and time formulas over values of these clocks to express timing constraints of complex systems. We present also some algorithms to extract time annotations that adorn sequence diagrams and transform them into timing constraints in our timed graphs. Obviously, this approach alleviates more the hard task of consistency checking between UML diagrams, specifically interaction diagrams with regards to state diagrams. Timeliness and performance analysis of timed graphs related to sequence diagrams could take advantages of works on model checking of timed automata.

1 Introduction

Scenarios-based specifications have become increasingly accepted as a means of requirements elicitation for concurrent systems such as telecommunications software. Indeed scenarios describe in an intuitive and visual way how system components and users interact in order to provide system level functionality. They are also used during the more detailed design phase where the precise inter-process communication must be specified according to formal protocols [3, 4, 7, 10, 12, 8].

The Unified Modeling language (UML [8]) which is an OMG standard and multi-paradigm language for description of various aspects of complex systems, adopted early this kind of visual and flexible notations for expressing the interactions between system components and their relationships, including the messages that may be dispatched among them. More precisely, UML contains 3 kinds of these interactions diagrams: sequence diagrams, communication diagrams and timing diagrams [8].

Recently, many established features of MSC (Message Sequence Charts) [5] have been integrated into the version 2.0 of UML [8], namely the interaction operators among fragments and the adoption of partial order among interaction events rather than the related messages.

Each interaction fragment alone is a partial view of the system behavior but when combined all together by means of the new interaction operators, interactions provide relatively a whole system description.

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 259 – 274, 2006.

However in spite of the expressiveness and precise syntactic aspects of UML notations, their semantics remain described in natural language with sometimes OCL formulas. Accordingly, to use automated tools for analysis, simulation and verification of parts of produced UML models, UML diagrams should be given a precise and formal semantics by means of rigorous mathematical formalisms [12, 4].

In this context, this paper presents a new formal approach to defining branching time semantics for UML Sequences Diagrams in denotational style. Our approach deals with the partial order and yields lattice-like graphs that specify faithfully the intended behaviors by recording both traces of all interaction components together with branching bifurcations. We provide our mathematical structure with few generalized algebraic operations making it easy to give formal definitions of interaction operators in a compositional manner. Next we extend our formalism with logical clocks and time formulas over these clocks to express timing constraints of complex systems. Some given algorithms show how to extract time annotations of sequence diagrams and transform them into timing constraints in our timed graphs.

Obviously, this approach alleviates more the hard task of consistency checking between UML diagrams, specifically interaction diagrams with regards to state diagrams. Timeliness and performance analysis of timed graphs related to sequence diagrams could take advantage of works on model checking of timed automata [1].

The paper is structured as follows: The next section shows the basic features of UML Sequences Diagrams (DS) and in section 3 we present our formal model and its algebraic operations. Then in section 4, the semantics of DS are given in compositional style by combining the denotations of interaction fragments using our algebraic operations and section 5 presents a temporal enhancement of our graphs with logical clocks and temporal formulas and then we give the method to extract into our timed graphs the timing constraints from time annotations on sequence diagrams. Finally last sections compare related work with ours and give concluding remarks.

2 Interactions and Sequences Diagrams

The notation for an interaction in a sequence diagram is a solid-outline rectangle of which upper left corner contains a pentagon. Inside this pentagon, the keyword **sd** is written followed by the interaction name and parameters [8].

In a sequence diagram (Fig.1), participants (components, objects …) that participate in the interaction are located at the top of the diagram across the horizontal axis. From each component shown using a rectangle, a lifeline is drawn to the bottom and the dispatching of every message is depicted by a horizontal arc going from the sender component to the receiver one. These messages are ordered from top to bottom so that the control flow over time is shown in a clear manner. Each message is defined by two events: message emission and message reception and events situated on the same lifeline are ordered from top to down [8].

Accordingly a message defines a particular communication among communicating entities. This communication can be “raising a signal”, “invoking an operation”, “creating” or “destroying an instance”. The message specifies not only the kind of communication but also the sender and the receiver. The various kinds of communication

involved in distributed systems are considered in UML sequence diagrams. Hence messages may be either synchronous or asynchronous [8].

Basic interaction fragment only represents finite behaviors without branching (when executing a sequence diagram, the only branching is due to interleaving of concurrent events), but these can be composed to obtain more complete descriptions. Basic interaction fragments can be composed in a composite interaction fragment called combined interaction or combined fragment using a set of operators called interaction operators. The unary operators are OPT, LOOP, BREAK and NEG. The others have more than one operand, such as ALT, PAR, and SEQ. Recurrently the combined fragments can be combined themselves together until obtaining a more complete diagram sequence [8] (see fig.1).

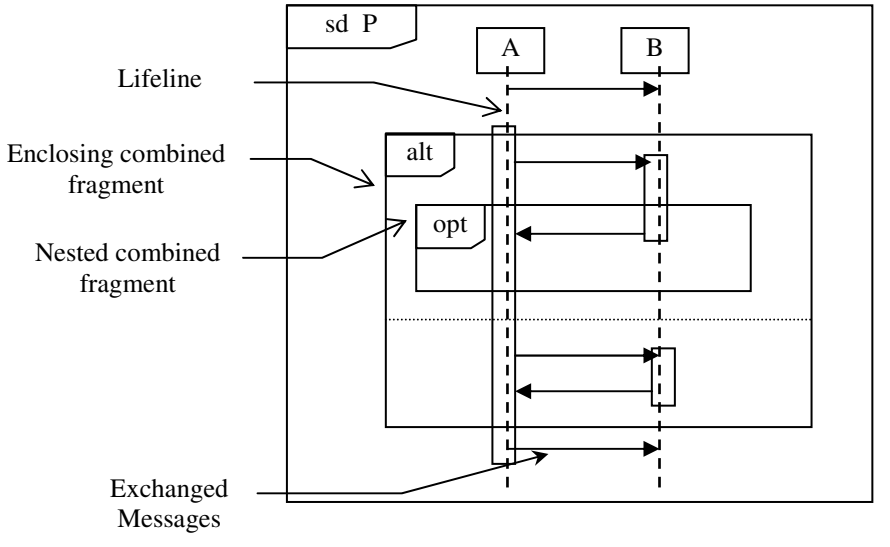

Fig. 1. Sequence Diagram.

The notation for a combined fragment in a sequence diagram is a solid-outline rectangle. The operator is in a pentagon at the upper left corner of the rectangle.

The operands of a combined fragment are shown by tiling the graph region of the combined fragment using dashed horizontal lines to divide it into regions corresponding to the operands.

Finally interactions in sequence diagrams are considered as collections of events instead of ordered collections of messages as in UML 1.x. These stimuli are partially ordered based on which execution thread they belong to. Within each thread the stimuli are sent in a sequential order while stimuli of different threads may be sent in parallel or in an arbitrary order.

Notations. Let $\sum$ be a vocabulary of symbols. $\sum^*$ is the set of all finite words over $\sum$ including the empty word ε. Let $w \in \sum^*$, $|w|$ denotes the length of w and w(n) denotes the n^{th} symbol of w. If u, $v \in \sum^*$, u.v denotes the concatenation of u and v.

3 Formal Model for Interaction Behavior

We present below the mathematical model used for the definition of the branching time semantics of sequence diagrams. This model is a *lattice-like* graph which records faithfully from sequences diagrams the intended traces of events together with the choices possibilities. Moreover it preserves the partial order among events in such a way that structure may contain diamond-shaped parts.

$G = <O, \Sigma, <_\Sigma, S, s_0, T>$ where :

- O is the set of participants involved in an interaction.
- Σ is the set of events occurrences. Note that Σ contains an unobservable event $\tau \neq \varepsilon$ modeling change of control flow. This event τ may be also adorned with a guard.
- $<_\Sigma \subset \Sigma \times \Sigma$, is a set of pairs of events occurrences where each one represents a binary relation between two occurrences to describe that one event must occur before the other in a valid trace. This mechanism provides a partial order on events occurrences so that the set of possible sequences is more restricted.
- $S = \{ s_k : O \rightarrow \Sigma^*, k \in \aleph \}$
  $= \{ s_k : o_i \rightarrow w \in \Sigma^* / \forall n \leq |w|, \forall m \leq n : (w(n) , w(m)) \notin <_\Sigma \}$

Every mapping s_k from S assigns to each participant some word (trace of events occurrences) at some point k of its evolution such that the binary relation $<_\Sigma$ among events remains preserved. Hence, these mappings constitute the nodes of the graph.

- $s_0 \in S$ represents the initial node where no event is recorded. $\forall o_i \in O, s_0(o_i) = \varepsilon$.
- $T : S \times \Sigma \rightarrow S$
  $(s_i, e) \rightarrow s_j$

Each transition records any occurring event different from τ onto the trace of the relating object. For more convenience, we write s_i |----e-----s_j.

If $e = \tau$ then $\forall o \in O: s_j(o) = s_i(o)$

If $e \neq \tau$ then there exits exactly one object o where:

$e \in$ lifeline(o) $\wedge\ s_j(o) = s_i(o).e \wedge \forall o' \in O, \forall e' \in s_i(o') : (e,e') \notin <_\Sigma$.

(e should never occur if it precedes any other recorded event via the partial order)

$\forall o' \neq o: s_j(o') = s_i(o')$ (s_j does not record e onto o' trace if o' $\neq$ lifeline (e)).

We prefer call final nodes leaf nodes rather than acceptance nodes because sequence diagrams are only some pieces of the expected behavior. So any recorded trace is only a prefix of some whole traces we can only compute from State Diagrams.

Below we define two binary and one unary algebraic operations on these kinds of graphs. These operations are generalized making it possible to define the formal semantics of interaction operators on interaction fragments in a compositional style.

3.1 Choice Operation

This operation achieves an adjunct of graphs. Choice is made between them via internal τ-actions. Let G_1, G_2 be two graphs where:

$G_1 = <O^1, \Sigma^1, <_\Sigma^1, S^1, s_0^1, T^1>$, $G_2 = <O^2, \Sigma^2, <_\Sigma^2, S^2, s_0^2, T^2>$ Such that $O^1 = O^2$

$G_1 \oplus G_2 = <O, \Sigma, <_\Sigma, S, s_0, T>$ where :

- $O = O^1 = O^2$.
- $\Sigma = \Sigma^1 \cup \Sigma^2$.
- $<_\Sigma = <_\Sigma^1 \cup <_\Sigma^2$

- $S = S^1 \cup S^2 \cup \{s_0\}$ where $s_0 \in S$ is new initial node.
- $T = T^1 \cup T^2 \cup T'$ where $T' = \{s_0 \text{ |---}\tau\text{--- } s_0^1, s_0 \text{ |---}\tau\text{--- } s_0^2\}$

3.2 Parameterized Cartesian Product

This product achieves merging of all pairs of traces from the two graphs but in such a way the partial order among events remain preserved. Whenever we try concatenate two traces, we should check that none of events occurrences of the second trace is ordered before an event from the previous trace.

Let G_1, G_2 be two graphs where:

$G_1 = <O^1,\Sigma^1, <_\Sigma^1, S^1, s_0^1, T^1>$, $G_2 = <O^2,\Sigma^2, <_\Sigma^2, S^2, s_0^2, T^2>$

$G_1 \otimes_{Prior} G_2 = <O, \Sigma, <_\Sigma, S, s_0, T>$ where :

- $O = O^1 \cup O^2$. - $\Sigma = \Sigma^1 \cup \Sigma^2$ - $<_\Sigma = <_\Sigma^1 \cup <_\Sigma^2 \cup$ Prior

$Prior \subseteq (\Sigma^1 x\, \Sigma^2) \cup (\Sigma^2 x \Sigma^1)$ is a subset of new particular order relations among events.

- $S = PRUNE\ (\ (S^1 \otimes S^2) \cup (S^2 \otimes S^1)\ )$.

$$s_k \in S : o \rightarrow s_k(o) = \begin{cases} s_i^1(o) \otimes s_j^2(o) & \text{where } s_i^1 \in S^1 \text{ and } s_j^2 \in S^2 \\ s_i^2(o) \otimes s_j^1(o) & \text{where } s_i^2 \in S^2 \text{ and } s_j^1 \in S^1 \end{cases}$$

$$s_i^1(o) \otimes s_j^2(o) = \begin{cases} s_i^1(o).s_j^2(o) \text{ such that } o \in O^1 \cap O^2 \wedge \forall e \in s_i^1(o),\ \forall e' \in s_j^2(o): (e',e) \notin <_\Sigma \\ s_i^1(o) \text{ if } o \notin O^2 \\ \varepsilon \text{ otherwise.} \end{cases}$$

$$s_i^2(o) \otimes s_j^1(o) = \begin{cases} s_i^2(o).s_j^1(o) \text{ such that } o \in O^1 \cap O^2 \wedge \forall e \in s_i^2(o),\ \forall e' \in s_j^1(o): (e',e) \notin <_\Sigma \\ s_i^2(o) \text{ if } o \notin O^1 \\ \varepsilon \text{ otherwise.} \end{cases}$$

The function PRUNE removes all unreachable nodes from the initial node through T.

- $s_0 = s_0^1 \otimes s_0^2 = s_0^2 \otimes s_0^1$ (hence $\forall o \in O: s_0(o) = \varepsilon$).

- $T : S \times \Sigma \rightarrow S$

$T = \{(s_k,e,s_{k'})/s_k = s_i^1 \otimes s_j^2,\ s_{k'} = s_m^1 \otimes s_n^2,\ \exists o \in O: (s_i^1(o),e,s_m^1(o)) \in T^1 \vee (s_j^2(o),e,s_n^2(o)) \in T^2\}$
$\cup \{(s_k,e,s_{k'})/s_k = s_i^2 \otimes s_j^1,\ s_{k'} = s_m^2 \otimes s_n^1,\ \exists o \in O: (s_i^2(o),e,s_m^2(o)) \in T^2 \vee (s_j^1(o),e,s_n^1(o)) \in T^1\}$

3.3 Star Operation

This operation adds to the graph new τ-transitions outgoing from leaf nodes to the initial node. Furthermore it adds a new empty node s_ε connected to the initial node by a τ-transition (see figure 2). Let G_1 be the starting graph $G_1 = <O^1,\Sigma^1, <_\Sigma^1, S^1, s_0^1, T^1>$

STAR $(G) = <O, \Sigma, <_\Sigma, S, s_0, T>$ where :

- $O = O^1$, $\Sigma = \Sigma^1$, $<_\Sigma = <_\Sigma^1$, $S = S' \cup S''$
- $S' = \{s_k : O \rightarrow \Sigma^*, k \in \aleph\}$ where for all k we have :

$$s_k : o \rightarrow s_k(o) = \begin{cases} (s_k^1(o))^+ & \text{if } s_k \in LEAF(S^1) \\ s_k^1(o) & \text{otherwise.} \end{cases}$$

- $S'' = \{s_\varepsilon : O \rightarrow \{\varepsilon\}\}$

The sole node s_ε records empty traces for all objects.

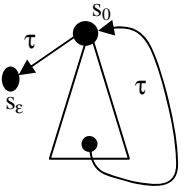

Fig. 2. Star operation

- $s_0 = s_0^1$.
- $T = T^1 \cup T' \cup T''$ / $T' = \{\forall s_F \in LEAF(S^1): s_F \text{ |---}\tau\text{--- } s_0^1\}$ where
  $LEAF(S) = \{s \in S / NOT (\exists s' \in S, e \in \sum : s \text{ |---}e\text{--- } s')\}$

$T'' = \{ s_0^1 \text{ |---}\tau\text{--- } s_\varepsilon\}$

Property. The two operations $\otimes$ and $\oplus$ on graphs are associative.

Lemma 1. let G be the graph $<O, \Sigma, <_\Sigma, S, s_0, T>$
$\forall s_k \in S, \forall o \in O: u=s_k(o) \Rightarrow (\forall i, j \in \aleph : i , j \leq |u| \wedge (u(i),u(j)) \in <_\Sigma) \Rightarrow i<j$.

Proof. Definitions of S and T compel event occurrences concerned by $<_\Sigma$ to occur in a way so that the partial order remains preserved. The other events may appear in any order in the sequence.

Definition 1. Let u and w be two sequences from $s_k(o)$ ($o \in O$, $s_k \in S$)
u and w are equivalent (we write $u \approx w$) if and only if $\forall a \in \sum: a \in u \Leftrightarrow a \in w$.
The precedence relation is preserved for ordered events in both u and w.

Definition 2. Let s_i and s_j be two nodes from S
s_i and s_j are equivalent (we write $s_i \approx s_j$) if and only if $\forall o \in O: s_i(o) \approx s_j(o)$.

Definition 3. Let G be a graph = $<O, \Sigma, <_\Sigma, S, s_0, T>$. A reduced graph (automaton) may be obtained from the graph G by reducing the equivalent nodes into equivalence class of nodes as follows: $G = <O, \Sigma, <_\Sigma, S', s_0, T'>$

$S' = \{[s_k] / s_k \in S\}$ where $[s_k] = \{s_i \in S / s_i \approx s_k\}$.

$T' = \{ [s_k] \text{ |----e------}[s_{k'}] / \exists s_i \in [s_k], \exists s_j \in [s_{k'}], \exists (s_i \text{ |----e-----}sj) \in T\}$

Remark1. On the other hand side, we can unfold our graph (namely cycles and diamond shapes) in order to obtain the equivalent transition system.

3.4 Handling of Synchronous Messages

Although the subset "Prior" is used particularly to handle specific features of interaction operators used among combined fragments (as explained later), we can also use it to handle synchronous messages when assembling jointly many lifelines in one interaction fragment or when combining many sequence diagrams by means of interaction operators. We have only to add into the partial order subset "Prior" other general orderings with regards to send and receive events of those specific messages.

Let M be the set of synchronous messages between two combined fragments SD_i and SD_j (which could be only lifelines of participants).

"Prior" is then increased with the set $\cup_{m \in M}(Prior_m)$ where :

$Prior_m = \{(m!,e) \quad / m! \in \sum_j \wedge \exists e \in \sum_i : (m?,e) \in <_{\Sigma i}\}$
$\cup \{(m?,e') / m? \in \sum_i \wedge \exists e' \in \sum_j : (m!,e') \in <_{\Sigma j}\}$

This means that once the send event (m!) occurs the executing thread will stop until the receive event (m?) occurs on the other lifeline thanks to its precedence level against the successive events of the send event. Similarly, if we observe first a receive

event on the second participant lifeline, the related thread should stop until the send event occurs on the first participant. Note that only related threads to these events should synchronize and other concurrent threads could continue performing parallel activities and generating others events.

4 Formal Semantics of Interaction Fragments and Operators

4.1 Lifeline of a Participant

We associate to each interaction fragment X a related graph denoted |[X]|.
Let P be a participant in some interaction. Its graph is $|[P]|=<O,\Sigma, <_\Sigma, S, s_0, T>$ where:

- $O = \{P\}$ is a singleton set consisting in the only one participant P.
- $\Sigma = \{ e / \exists m: e = Receive(m) \wedge Receiver(m)=P \vee e = Send(m) \wedge Sender(m)=P\}$
- $<_\Sigma = \{(e,e') \in \Sigma x \Sigma$ / the event occurrence e occurs before e' on the lifeline(P)$\}$

Frequently the order on a same lifeline is total i.e. if we take two event occurrences e, e' on the same lifeline then e<e' or e'<e. But if the lifeline of P contains a coregion area then the order of event occurrences on this part is insignificant.

- $S = \{s_k : O \rightarrow \Sigma^*, k \in \aleph \}$
  $= \{s_k : P \rightarrow w / \forall n \leq |w|, \forall m \leq n : (w(n) , w(m)) \notin <_\Sigma \}$
- $s_0(P) = \varepsilon$.
- $T = \{(s_i,e,s_j) / s_j(P) = s_i(P).e \ \wedge e \in \sum\}$

Thus each transition yields a new trace onto the target node by adding its labeling event occurrence to the previous trace recorded in the source node of the transition.

In the following example (fig.3), we use notational shorthand called "coregion area" for combined fragments where the order of events occurrences on the lifeline is insignificant.

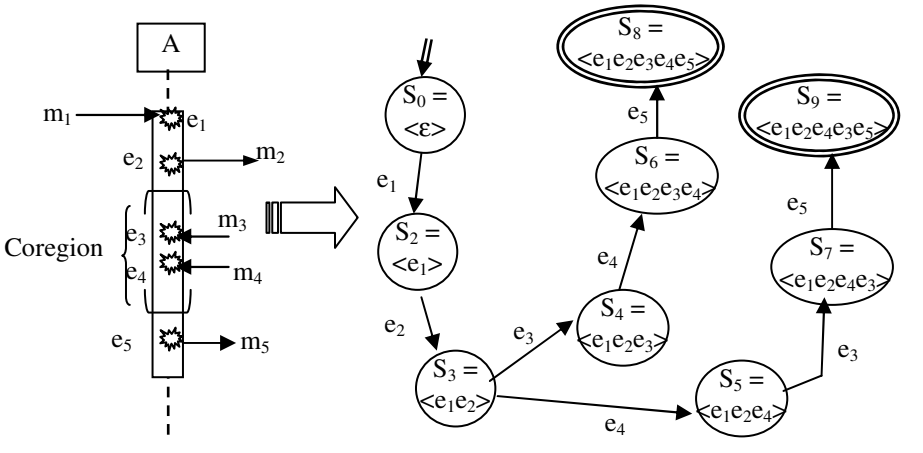

Fig. 3. The graph related to a life-line of one participant ($s_6 \approx s_7$. $s_8 \approx s_9$).e1 = m1?, e2 = m2!, e3=m3?, e4 = m4?, e5=m5!

4.2 Basic Interaction Fragment

Recall that a basic interaction fragment is a piece of an interaction which involves many participants without using any interaction operator.

Let DS be a basic interaction between two participants P_1 and P_2. Herein $O^1 \cap O^2 = \varnothing$. The graph related to DS is obtained by a parallel merge of the graphs relating to the participants lifelines with respect to the partial order between send and receive events.

$|[DS]| = |[P1]| \otimes_{Prior} |[P2]|$ where :

$Prior = \{(e,e') \in (\Sigma^1 x \Sigma^2) \cup (\Sigma^2 x \Sigma^1) \ / \ \exists$ message m : e=send(m) $\wedge$ e'=receive(m) $\}$

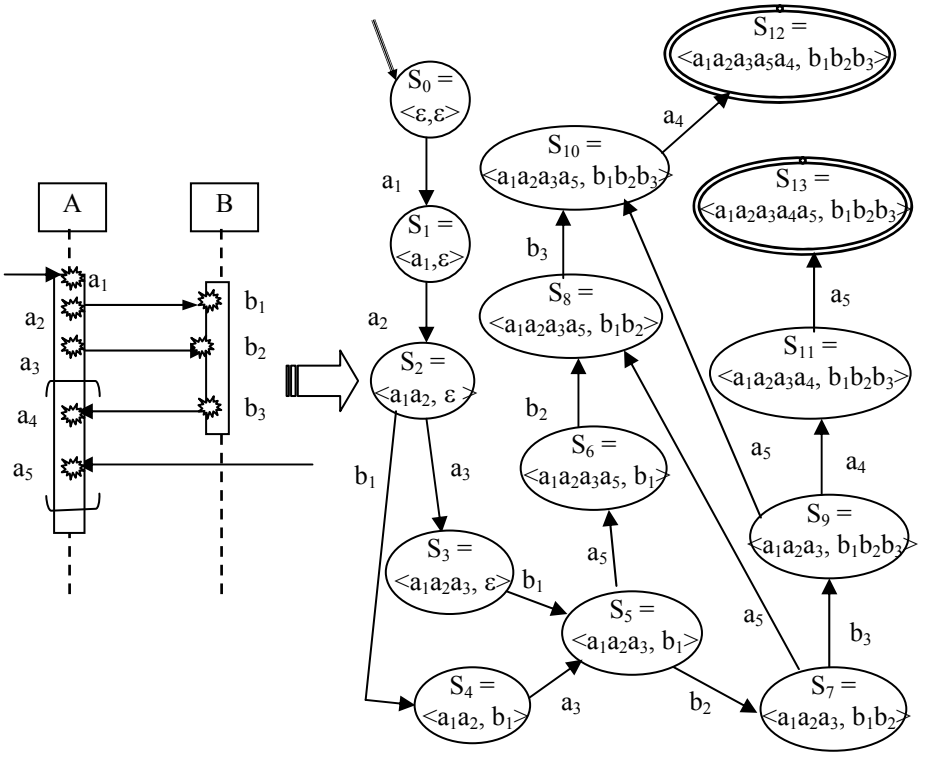

Fig. 4. the graph related to an interaction fragment with two participants

4.3 Choice Operator ALT

Let DS be an interaction fragment combined from two interaction fragments DS_1 and DS_2 by means of the choice operator **ALT**. This operator indicates that the resulting fragment represent a choice of behavior. At most one the operands will be chosen.

Formally, $|[DS_1$ **ALT** $DS_2]| = |[DS_1]| \oplus |[DS_2]|$ (see fig.5)

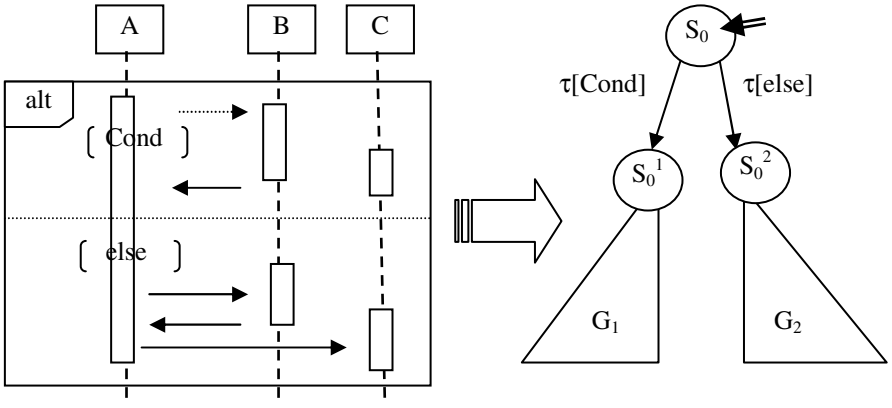

Fig. 5. The graph related to an interaction fragment with ALT operator

4.4 Operator BREAK

Even though **BREAK** is a unary operator which operand is a nested fragment in an enclosing interaction fragment, this operator can reduce to an interaction operation **ALT** between the nested fragment and the remainder of the enclosing interaction fragment.

4.5 Operator OPT

Let DS' be a new interaction fragment obtained by applying the operator **OPT** on another interaction fragment DS. The operator **OPT** designates that the resulting fragment represents a choice of behavior where either the sole operand happens or nothing happens. Formally, this means:

$$|[\ DS']| = |[\mathbf{OPT}(DS)]| = |[DS\ \mathbf{ALT}\ DS_{\varnothing}]| = |[DS]| \oplus |[DS_{\varnothing}]|$$

The empty interaction fragment $DS_{\varnothing}$ is mapped into an empty graph as follows:

$|[DS_{\varnothing}]| = <O,\varnothing,\varnothing,\{s_0\}, s_0, \varnothing>$ where O is the same collection of participants in DS and s_0: $O \rightarrow \{\varepsilon\}$ associates to each participant in O an empty sequence of events.

4.6 Operator PAR

Let DS be interaction fragment combined from two interaction fragments DS_1 and DS_2 by means of the parallel operator PAR. We realize here an interleaving between all sequences occurring in diagrams (fig.6) without adding further orderings.

$$|[DS_1\ \mathbf{PAR}\ DS_2]| = |[DS_1]| \otimes_{Prior} |[DS_2]| \text{ where Prior} = \varnothing.$$

4.7 Operator of Strict Sequencing

Let DS be an interaction fragment combined from two interaction fragments DS_1 and DS_2 by means of the strict sequencing operator **SEQs**.

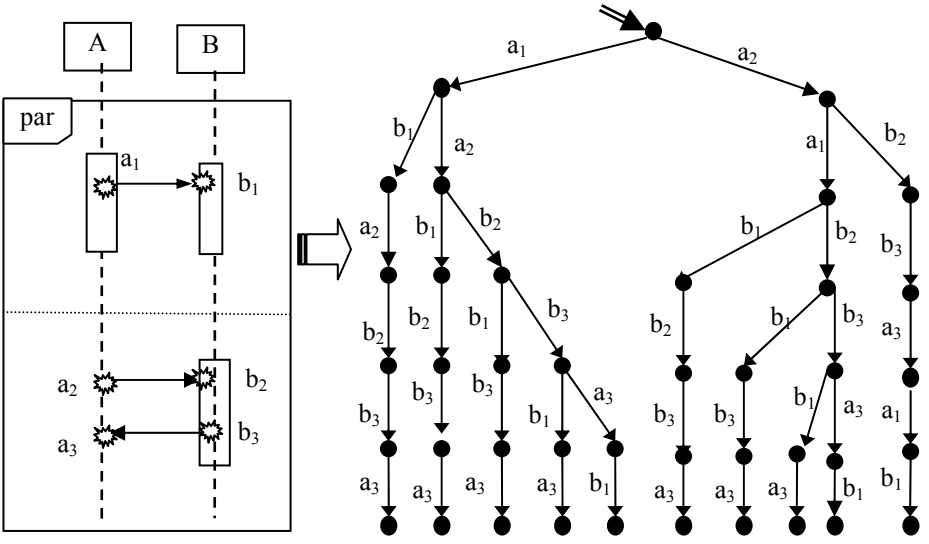

Fig. 6. The graph related to an interaction fragment with PAR operator

Similarly to sequential composition in process algebra, the semantics of strict sequencing defines a strict ordering of the operands on the first level. So the former operand should first be carried out entirely. After that only the second one is executed. Therefore all events occurrences of the first interaction fragment are granted more priority over those of the second one.

$$|[DS_1 \textbf{ SEQs } DS_2]| = |[DS_1]| \otimes_{Prior} |[DS_2]| \text{ where Prior} = \Sigma^1 \text{ x } \Sigma^2.$$

Herein we carry out concatenations between paths belonging respectively to the two diagrams DS_1 and DS_2 such that events occurrences from the first diagram always occur before those of the second diagram.

4.8 Operator of Weak Sequencing

Let DS be an interaction fragment combined from two interaction fragments DS_1 and DS_2 by means of the weak sequencing operator **SEQw**.
The weak sequencing expresses three properties [8]:

- The ordering of events occurrences within each of the operands are maintained in the result.
- Occurrence specifications on different lifelines from different operands may come in any order.
- Occurrence specifications on the same lifeline from different operands are ordered such that an event occurrence of the 1st operand comes before that of the 2nd one.

Hence the weak sequencing reduces to a parallel merge between events occurrences on different lifelines but restricted by strict sequencings among events

occurrences belonging to same lifelines (thanks to Prior set of added precedence relations). Formally, $|[DS_1 \textbf{ SEQw } DS_2]| = |[DS_1]| \otimes_{Prior} |[DS_2]|$

where Prior={(e,e')$\in \Sigma^1 x \Sigma^2$ / e,e' belong both to the same lifeline(o)}.

4.9 Operator LOOP

Let DS be a combined fragment from an interaction fragment DS_1 by means of the loop operator **LOOP** parameterized by a guard G given as un integer $\in$ {min ..max}.

The loop operator would be repeated a given number of times as long as the guard is fulfilled. |[LOOP(G, DS)]| = (|[(DS **SEQs** LOOP(G-1, DS)]|

However this solution does not pay attention when iterating to the evaluation event of the loop guard. So we have to add τ-transitions to record this internal choice.

|[**LOOP**(G, DS)]| = (|[(DS **SEQs** DS_τ) **SEQs LOOP**(G-1, DS)]|

The graph of τ-interaction fragment DS_τ is: $|[DS_\tau]| = <O,\{\tau\},\varnothing,\{s_0,s_1\}, s_0, \{s_0|\text{--}\tau\text{---}s_1\}>$

When the number of iterations is undefined (max = ∞), the correct solution consists in using the star operation on traces with adorning loop transitions with τ which models guard evaluation (see fig. 2). So the related graph should be.

|[**LOOP**(G,DS)]| = STAR (|[DS]|)

Remark2. we prefer use strict sequencing rather than weak one to avoid a pathological case of divergence in loop combination when using asynchronous communication

Remark3. All above rules can be used to handle the operator **NEG** in order to build the graph containing invalid traces of events with recording all branching choices.

5 Extraction of Timing Information

The sequence diagram in figure 7 shows how time and timing notations may be applied to describe time observation and timing constraints [8]. The "User" sends a message "Code" and its duration is measured. The "ACSystem" will send two messages back to the "User". "CardOut" is constrained to last between 0 and 13 time units. Furthermore the interval between sending of Code and the reception of "OK" is constrained to last between d and 3*d where d is the measured duration of the "Code" signal. We also notice the observation of the time point t at the sending of "OK" and how this is used to constrain the time point of the reception of "CardOut".

Our approach consists in extracting time formulas over logical clocks from the time annotations in sequence diagrams. Then we adorn related nodes and transitions in our graph by these timing conditions in a similar way to timed automata [1].

Definition 4 (timing constraint). Let H be a finite set of clocks ranging over $\Re>0$ (set of non negative real numbers). The set $\Psi(H)$ of timing constraints on H is defined by the following syntax: $\psi ::= \text{true} \mid x \langle\langle c \mid x - y \langle\langle c \mid \text{not } \psi \mid \psi \wedge \psi$

where x, y $\in$ H, c $\in \Re$ (Integers) and $\langle\langle \in \{<, \leq\}$.

Other assertions such as, x>3, 2$\leq$x<y+5, $\psi \vee \psi$' can be defined as abbreviations.

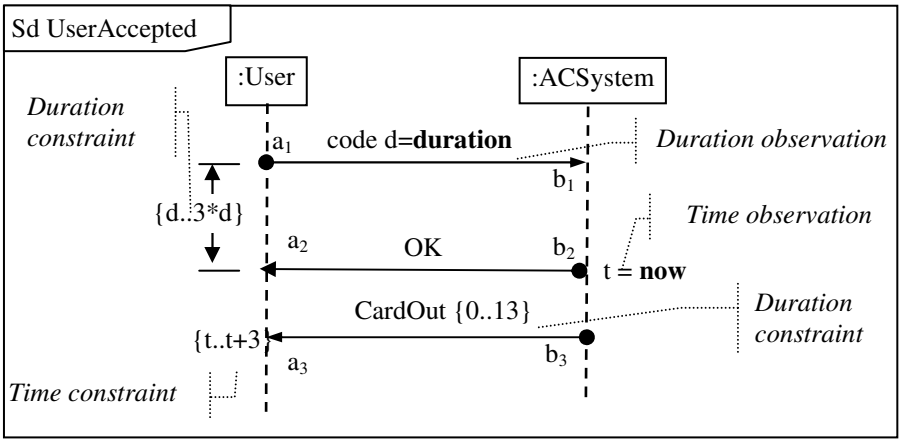

Fig. 7. Sequence Diagram with timing concepts

5.1 Enhancing Graphs with Timing Constrains

We add two mappings δ1, δ2 as follows:

$\delta_1 : S \rightarrow \Psi(H)$
$\delta_2 : T \rightarrow 2^H$,

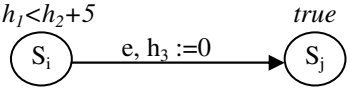

Fig. 8. Timed graphs

The first mapping δ_1 assigns to each node a condition called activity *condition* which may be *true*. The second mapping δ_2 associates with each transition a set of clocks initializations which may be empty.

The behavior of the new timed graph becomes as follows:

The control could stay in a node s_i (Fig.8) while the constraint $\delta_1(s)$ is fulfilled but once $\delta_1(s)$ becomes false we should leave s_i by execution of an instantaneous event occurrence e (an event occurs with no duration [8]). It's obvious that the control could stay indefinitely in s_j (Fig.8) if its activity condition is *true*.

When a transition t occurs, all clocks ($h_i \in \delta_2(t)$) are reset to zero. So these clocks start measuring time progress since this point but may be used later at different instants.

5.2 Extracting Time Constrains from Sequence Diagrams

Time observations, timing constraints are related to points on the lifelines of the sequence diagram. These points are the instances at which send or receive events occur. Likewise, duration observations or constraints are related to messages, each one of them is related to two events on the same lifeline or on two different lifelines.

The main idea of our approach to handling time constraint is to generate a logical clock "h" at any related time observation point "t". Any outgoing arc from this point will be adorned with initialization of the clock h making it possible to count time progress from this starting point.

For a time constraint of the form "t+a...t+b", we search out in our graph the set of nodes of which outgoing transitions are labeled with the event related to this constraint. Every such a node should receive a timing constraint of the form a≤h≤b.

```
Algorithm Extract_time & Duration_constraints
   Input  SD : Sequence Diagram;  G : Graph
   Output G' : a timed graph.

   H := ∅;
   For each s∈S do δ1(s) = true;

   For each time observation "t" at an event occurrence e
   Do {
        Generate a new clock h;  H := H ∪ {h};
        // h measures time progress since the observation point
        Find out the set A of transitions labeled with e;
        For each t∈A do δ2(t) = δ2(t) ∪(h,0);

        For each time constraint c of the form {a...b} on event
        occurrence e';
        Do {
             Find out the set N of nodes which outgoing transi-
             tions are labeled with e';
             For each s∈N do δ1(s) = δ1(s) ∧ h≥a ∧ h ≤b;
           }
      }
```

Likewise, for handling duration constraint, we generate two logical clocks; "h_1" at start point and "h_2" at final point related to duration observation "d". Any outgoing arc from these points will be tagged with initializations of related clocks so that the difference between their values (h_1-h_2) gives later the duration between the two events.

For any duration constraint of the form "a(d)..b(d)" between two events e and e', we add in a similar way a clock h_3 related to the first event e for counting the time progress since this first point. Next we search out in our graph the set of nodes of which outgoing transitions are labeled with the second event e'. Every such a node should then receive a timing constraint of the form $a(h_1\text{-}h_2) \leq h_3 \leq b(h_1\text{-}h_2)$.

```
   For each duration observation d on a message m
   Do {
        Let e be the event occurrence related to sending (m);
        Let e' be the event occurrence related to receiving (m);
        Generate two new clocks h1 and h2;  H := H ∪ {h1, h2};
        // h1 starts at the sending moment of m.
        Search out the set A of transitions labeled with e;
        For each t∈A do δ2(t) = δ2(t) ∪(h1,0) ;
        // h2 starts at the receiving moment of m.
        Find out the set B of transitions labeled with e';
        For each t∈A do δ2(t) = δ2(t) ∪(h2,0);

        For each duration constraint c of the form {a(d)...b(d)}
        between two events (e",e"') or on a message m'
        Do {
```

```
            Let e″ be the first event occurrence or the sending
            event of m';
            Let e″' be the second event occurrence or the re-
            ceiving event of m';
            Generate a new clock h3;   H := H ∪ {h3};
            // h3 measures time since the occurrence of e″
            Find out the set A of transitions labeled with e″;
            For each t∈A do δ2(t) = δ2(t) ∪(h3,0) ;
            Find out the set N of nodes which outgoing transi-
            tions are labeled with e″'
            For each s∈N do δ1(s)=δ1(s)∧ h3≥a(h1-h2)∧ h3≤b(h1-
            h2);
          }
    }

For each duration constraint c of the form {a...b} between
two events occurrences (e,e') or on a message m
Do {
     Let e be the first event occurrence or the sending event
     of m;
     Let e' be the second event occurrence or the receiving
     event of m;
     Generate a new clock h;    H := H ∪ {h};
     Find out the set A of transitions labeled with e;
     For each t∈A do δ2(t) = δ2(t) ∪(h,0) ;
     Find out the set N of nodes which outgoing transitions
     are labeled with e'
     For each s∈N do δ1(s) = δ1(s) ∧ h≥a ∧ h ≤b;
   }
```

At last we notice that the above approach can be also extended in straight way to handle other possible cases of time constraints.

The timed graph related to the diagram of fig.7 is the following:

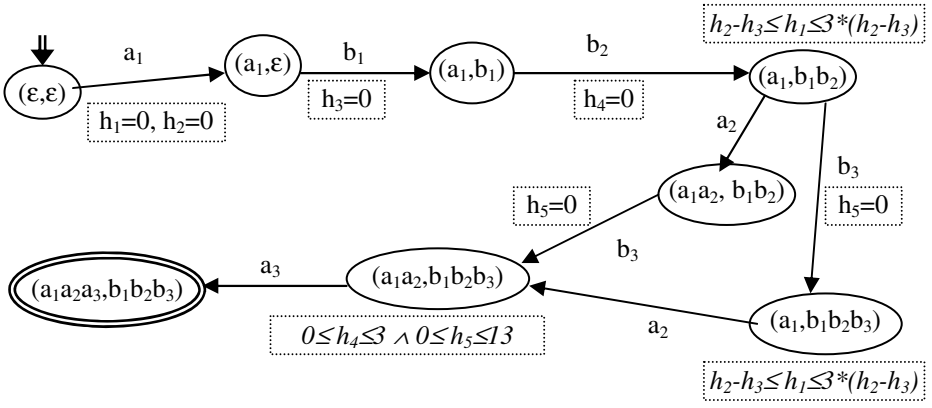

Fig. 9. The timed graph related to the sequence diagram depicted by figure 7

6 Related Work

Because of the widespread use of interaction diagrams in complex systems, many efforts have been made to give them formal meanings in order to allow systematic tool support during design, implementation and validation phases.

Besides the textual semantics given in UML specification document [8], many approaches [4], [10] present formal semantics of sequence diagrams (SD) where the runs are widely defined in terms of pairs of valid and invalid traces and do not record information about choice opportunities and coordination actions. Hence contrary to our approach, these linear time semantics are not enough faithful to allow complete consistencies checking over UML dynamic diagrams particularly in concurrent systems. Moreover, timing constraints are not handled within these models.

On the other hand side, some papers attempt to synthesize high level diagrams (StateCharts [12] [3], Petri nets [2]) from SD or MSC. However in our opinion as are assembly languages for high level programming ones, the sequence diagrams are less structured description languages in spite of the recent improvements. Furthermore the built high level models seem too unfolded or flattened and their high level syntactic constructs are not suitably used and may generate some inconsistencies with regards to the original sequence diagrams [12]. In this later work, the authors try to retrieve state diagrams of objects involved in interactions described by means of relatively simple sequence diagrams. The approach consists in deriving flat automata of some object from its lifelines in all interaction fragments and then combines them by means of simple interaction operators (choice, strict sequencing and loop). The other operators are discarded as the parallel operator so that the resulting automaton is flat and unfolded (without orthogonality) and may generate some irregular behaviors because of the removal of coordination information when extracting partial views.

An interesting work [3] considers good and bad interactions of reactive systems as safety and liveness properties that are described in terms of Büchi automata allowing refinement. SD traces become only prefixes of accepted infinite sequences and the various combinations between automata are not specified with regard to SD operators.

Another paper [7] uses process algebra terms to characterize the traces of scenario based specification that are defined by a causal ordering. It proves a canonical solution for correcting race conditions within the system behavior by weakening the causal relationship.

Note that many papers were proposed to overcome shortcomings of UML 1.x specification that relies on the ordering of messages instead of related actions. Hence authors of [2] and [9] proposed a formal semantics to the interaction diagrams of UML 1.x by the generation of an order relation that schedules the message emissions and receptions and can be automatically translated into a flattened Petri net or automata. Similarly, [6] presented a methodology to convert UML 1.x SD to a context-free grammar and applied parsing theory to locate non-determinism behavior. Additional information is discussed to attain deterministic behaviors for embedded systems modeling. Also, the approach of [11] proposes a formal semantics of UML 1.x sequence diagrams in terms of ordered hierarchical tree structure that represents the hierarchical relations among the messages (method invocations).

However, the new specification of UML 2.0 [8] adopts an ordering over events occurrences corresponding to sending and receiving of messages. Also high level

features of MSC [5] have been included in UML interactions allowing description of more complex behaviors. Moreover all the above works do not pay attention to time annotations on sequence diagrams.

7 Conclusion

In this paper, we have given a formal semantics for UML 2 sequence diagrams by using a faithfully branching time structure rather than traces. This model (a lattice-like graph) records both traces of all interaction components together with branching bifurcations and can be directly unfolded into a transition system capturing the intended behavior. The graphs related to interaction fragments are equipped with few generalized algebraic operations which help us define the formal semantics of all interaction operations in compositional manner. Moreover, we have proposed a method to extract time properties of UML interactions into time constraints we add to our graph in order to achieve timeliness and performance analysis.

Hence resulting graphs modeling valid and invalid behaviors would be compared to the state diagram to achieve semantically and temporal consistencies checking.

References

1. R. Alur, D. Dill. A theory of timed automata. Theorical Computer Science. 126 (1994) 183-235.
2. J. Cardoso, C. Sibertin-Blanc. An operational semantics for UML interaction: sequencing of actions and local control. European Journal of Automatised Systems. APII-JESA 36 P.1015-1028 (ISBN 2-7462-0573-4), Hermés-Lavoisier 2002.
3. R. Grosu and S.A. Smolka. Safety-Liveness Semantics for UML 2.0 sequence diagrams. In Proc. of ACSD'05, the 5th International Conference on Application of Concurrency to System Design, Saint-Malo, France. June 2005.
4. ∅. Haugen, K.E. Husa, R.K. Runde, K. Stølen. STAIRES towards formal design with sequence diagrams. Software & System Modeling, online first: 1-13, 2005.
5. ITU-T. Z.120. Message sequence charts (MSC), November 1999.
6. E. Latronico and P. Koopman, Representing Embeded System Sequence Diagrams as a Formal Language. In Proc. of UML'2001 Conference, Toronto Ontario, 3-5 Oct.2001.
7. B. Mitchell. Inherent Causal Orderings of Partial Order Scenarios. In proc. of International Colloquium on Theorical Aspects of Computing, Guiyang, China, (LNCS 3407) PP 114-129. September 2004.
8. OMG. Unified Modeling Language: Superstructure version 2.0, Final Adopted Specification. Object Management Group, 2004 Available from http:// www.omg.org.
9. C. Sibertin-Blanc, O. Tahir and J. Cardoso. Interpretation of UML sequence diagrams as causality flows. In Proc. of ISSADS'2005, (LNCS 3563), pp. 126-140. 2005.
10. H. Störrle. Trace semantics of interactions in UML 2 .0. Technical Report TR 0403, University of Munich, Germany. 09/2004.
11. Xiaosham Li , Zhiming-Liu and He Jifeng. A formal semantics of UML sequence Diagram. In Proc. of Australian Software Engineering Conference 2004, Australia. April 2004.
12. T. Ziadi, L. Hélouët, J-M. Jézéquel. Revisiting statechart synthesis with an Algebraic Approach. In proc. of International conference on Software Engineering (ICSE'04). 2004.