An Operational Semantics of UML2.X Sequence Diagrams for Distributed Systems

Fatma Dhaou, Inès Mouakher, Khaled Bsaïes, Christian Attiogbé

An Operational Semantics of UML2.X Sequence Diagrams for Distributed Systems

Fatma Dhaou *, Ines Mouakher, J. Christian Attiogbé, and Khaled Bsaies

University Tunis El Manar, LIPAH, Faculty of Sciences of Tunis Tunisia,
University of Nantes,LS2N, Nantes, France

Abstract. UML2.X sequence diagrams (SD) are equipped with high structures: the combined fragments (CF) that permit to model complex behaviours of systems. CF can be nested to allow more sophisticated behaviours, however they complicate the interpretation of the SD and the computation of precedence relations between the events. In a previous work, we proposed a causal semantics for UML2.X SD. It is based partial order theory, its well-defined relations allow the computation of all precedence relations for the events of UML2.X SD with nested CF. We considered the most popular CF of control-flow ALT, OPT, LOOP, SEQ allowing to model respectively alternative, optional, iterative and sequential behaviours. In this work, we improve that previous work to consider a PAR CF allowing to model parallel behaviours, and we propose an operational semantics that is based on the causal semantics. The proposed operational semantics is a substantial step towards the refinement checking and the analysis of some properties of SD.

Keywords: UML2.X Sequence Diagrams, Operational Semantics, Causal Semantics, Nested Combined Fragments

1 Introduction

Context. The speed of design, the intuition and the ease of graphical representation make UML2.X sequence diagrams (SD) a privileged language often used by the engineers in the software industries. Although the Object Management Group (OMG) [1] has defined an official standard semantics for UML2.X SD, some shortcomings still persist. For instance, we report that the definitions of the standard semantics are not well suited for an exhaustive computation of all possible traces of basic SD modelling the behaviours of distributed systems this is a shortcoming. Moreover, they are not formalized which yields, in some cases, to the ambiguities of interpretations.

Motivation. The defined rules by the OMG for deriving partial order of a given basic SD impose to order the events along each lifeline, even if they are received from independent lifelines, which do not allow the computation of all

* Please note that the LNCS Editorial assumes that all authors have used the western naming convention, with given names preceding surnames. This determines the structure of the names in the running heads and the author index.

possible valid behaviours. This leads to the emergence of unspecified behaviours in the implementation. Although we can add coregion operator and additional messages to establish the required order, however we obtain an overcrowded graphical representation that can lead to the interpretation ambiguities. With UML2.X, the combined fragments allow the modelling of several kind of behaviours. We focus especially on a subcategory of CF: ALT, OPT, LOOP, SEQ and PAR; they permit a compact syntactic representation of behaviours. In contrast, they cause challenges for the determination of precedence relations between the events. To compute traces for SD equipped with these CF, the OMG standard recommends to compute the traces of each components of the SD independently then the traces are composed by the WEAK SEQUENCING operator. This processing is equivalent in other approaches [2], [3], [4], [5] to the flattening of the SDs that are semantically equivalent. However, the benefits of the compact syntactic representation are lost.
Moreover, the ALT and the LOOP CF have a different meaning than in the structured programming languages; although, to ease the processing of these CF, the existing approaches [6], [5], [7], restrict their use by interpreting them in the same way. However, in the standard they have much more flexible interpretations allowing to model more complex behaviours; for instance the ALT CF is not equivalent to the $IF - Then - Else$ structure, and in the LOOP CF, weak sequencing between the iterations is applied, rather than strict sequencing, permitting the interleaving of the occurrence of the events of different iterations.

In the practical cases, CF can be nested to model more sophisticated behaviours. All the cited problems are increasing. In the standard semantics, the notion of nested CF is briefly mentioned. In literature, few works [7], [5], [6] deal with nested CF. In [6] the authors study the issues resulting of the nesting of some kinds of CF (different of those considered in this paper), and by limiting the nesting levels of CF [6], [5], or by proposing a complicated formalization very close to the target formalism [7].

Although the existing semantics that are proposed for UML2.X SD are various [3], [8], [9], [10], but they are usually based on the definitions of the standard semantics for the computation of traces of the SD, thus they are not suitable for SD modelling behaviours of distributed systems. These shortcomings have motivate our proposal for a causal semantics dedicated for UML2.X SD with nested CF that models behaviours of distributed systems. Most of the existing semantics of different kinds (denotational, operational, algebraic) are based on the definitions of the standard semantics for the computation of precedence relations between the events, hence they present the same shortcomings as the standard semantics. Defining an operational semantics for SD facilitates their operational analysis and permits a better understanding of the language.

Contribution. This paper extends our previous works [11], [12]; in [11] we have extended the semantics that is proposed for UML1.X SD [13]; we have proposed several formal rules, to compute directly the partial order between the events of SD with the most popular combined fragments (ALT, OPT, LOOP)

that are sequential, by processing the SD as a whole. In [12], we have extended the formalization to deal with the nesting of (ALT, OPT and LOOP) CF, and we have generalized the precedence relations of the causal semantics that suit for UML2.X SD modelling the behaviours of distributed systems and equipped with nested CF.
We now propose additional contributions that consist in covering an other important CF that is the PARALLEL[1] CF, and we propose an operational semantics permitting a better understanding of the behaviour of the SD by defining the rules of occurrences of the events.

Organization. The remainder of the article is structured as follows. In Section 2 and 3, we provide an overview on our previous work: we explain the formalization of UML2.X SD and the precedence relations of the causal semantics. Section 4 is devoted to the operational semantics. Before concluding in Section 6, we present some related works in Section 5.

2 Causal Semantics

To overcome the shortcomings of the standard semantics, we considered an existing semantics [13] that is suitable for basic SD modelling behaviours of distributed systems. Its rules take into account the independence of the components, (modelled by lifelines), involved in the interactions. Indeed, in contrast with the standard semantics that totally order the events on each lifeline even for the receiving events from independent lifelines, the causal semantics imposes slighter scheduling constraints on the behaviour of lifelines results in more expressive SDs, since each SD describes a larger number of acceptable behaviours. This larger expressive power facilitates the task of the designer since a great number of cases have to be considered, and permits to prevent the issue of the emergence of unspecified behaviours in the implementation. The causal semantics is founded on a partial order theory. Intuitively, the causal semantics [14] is based on the idea of ordering events if there is a logical reason to do so. We present the relations of the causal semantics as defined in [13] in informal way as follows.
Synchronization Relationship $<_{SYNC}$. Each message m is received only if it was sent previously.
Reception-Emission Relationship $<_{RE}$. Receiving a message causes the sending of the message that is directly consecutive to it.
Emission-Emission Relationship $<_{EE}$. If two messages are sent by the same lifeline their sending events are ordered.
Causal order Relation $<_{caus}$. This relation is defined as follows:

$$<_{caus}= (<_{SYNC} \bigcup <_{RE} \bigcup <_{EE})$$

The transitive closure of the relation $<_{caus}$ that we note $<^{+}_{caus}$ permits to obtain all the causal dependencies between the events of the SD. The event occurrence

[1] The parallelism is logic, which mean that two events occur in any order.

depends on the partial order relationship $<_{caus}$.

The causal semantics is mainly proposed for basics UML1.X SD modelling behaviours of distributed systems, and the application of its rules causes some inconsistencies (aberrant relations, deadlock and inadvertent triggers of some events [11]). Hence in our previous work [12], we proposed a new formalization of UML2.X SD with nested CF that is based on set theory and the tree structure. Then, based on this formalization, we proposed the extension of the causal semantics whose its relations permit the computation of precedence relations for each event that belong to an UML2.X SD with nested CF modelling behaviours of distributed systems.

3 Overview on Previous Extension of the Causal Semantics

3.1 FORMALIZATION OF UML2.X SD WITH NESTED CF

We consider a sub-set of SD containing combined fragment of control-flow ALT, OPT, LOOP and SEQ CF. The considered CF are sequential, and can be nested to model more sophisticated behaviours. We assume that the operands of the CF do not overlap, but can be nested. For the formalization of sequence diagrams equipped with nested CF, we choose, on the one hand, the set theory notations [2] that is a privileged way due to its several advantages. For instance, although it is founded on first order logic, it permits to manipulate objects of high order such as sets and relations of any depth (that is, sets and relations built themselves on sets and relations, and so on) [15]. On the other hand, we use the tree structure that is hierarchic by nature and it is convenient to capture the nested structure of SD, and allow to represent them in an intuitive way.

Sequence Diagram Definitions

Definition 1 ***(Sequence Diagram)***
A sequence diagram SD is a tuple
$SD : \langle L, M, EVT, FCT\_s, FCT\_r, FCT\_l, OP, F, <_{caus}, tree\_OP\rangle$ *where:*

- L *is a set of not empty lifelines, and* $card(L) \geq 2$,
- M *is a set of asynchronous messages which is well formed and not empty. The set* M *is well formed if every message is identified by a pair of events: a sent event and a received event,*
- $EVT = E\_s \cup E\_r$ *is a set of events such that* $card(EVT) \geq 2$ [3]*,* E_s *and* E_r *denotes respectively the set of sent events and the set of received events such that* $E\_s = \{!m \mid m \in M\}$[4] *and* $E\_r = \{?m \mid m \in M\}$[5]*, and* $E\_s \cap E\_r = \emptyset$,

[2] N.B we use the same set theory notation as those of Event-B method
[3] Cardinal of a set E
[4] $!m$ denote the sent event of the m message
[5] $?m$ denote the received of the m message

- *for a set of message M we define two bijective functions FCT_s and FCT_r that permit to associate to each message respectively one sent event and one received event: $FCT\_s : M \rightarrowtail\!\!\!\!\!\rightarrow E\_s$* [6]*, and $FCT\_r : M \rightarrowtail\!\!\!\!\!\rightarrow E\_r$*
- *$FCT\_l : EVT \twoheadrightarrow L$* [7] *a total surjective function that associates to each event one lifeline, the transmitter or the receiver,*
- *$F = \{F_1, F_2, ..., F_n\}$ is the set of n CF, where $F_i = \langle OP_i, operator_i, L_i \rangle$ is a CF that is identified by its operands, an operator, and the set of lifelines that are covered by it,*
- *$<_{caus} \subseteq EVT \leftrightarrow EVT$ denotes the partial order relationship,*
- *OP: the SD is considered as a set of operands,*
- *$tree\_OP$ is a partial function that allows to structure the SD in the form of a tree of operands.*

To obtain the local order within each lifeline noted $<_{SD,l}$, we project the causal order relation $<^{+}_{caus}$ [8] on the lifeline l.

Operands of CF An SD is abstracted as a tree of operands. Intuitively, a combined fragment will be viewed as an operator together with its operands; this will be detailed in the sequel. We consider the following CF SEQ, ALT, OPT and LOOP. The SD is represented as a set of operands. We associate a label to each operand. Two operands with the same index i belong to the same combined fragment: it's the case of the operands of an ALT and PAR CF for instance, in Fig.1, OP_{21}, OP_{22} and OP_{23} belong to the same CF ALT.

The whole SD is transformed to a root operand that we note OP_{00}; the set OP is defined as $(\bigcup_{i=\{1..n\}} OP_i) \cup \{OP_{00}\}$; where n is the number of operands of the considered SD. Each operand in an SD has a weight. For instance, each operand of SEQ, ALT or OPT CF has a weight equal to 1; an operand of a LOOP CF has a weight equal to a value max, which is the maximum number of iterations of the considered LOOP CF. We assume that each operand of a CF has only one first event. The first events of the different operands of a same CF do not belong necessarily to the same lifeline, since some of them came from lower level when we built the tree.

The general definition of an operand in a combined fragment is given as follows.

Definition 2 ***(Operand in combined fragment)***
We define a set of operands OP_i in a CF F_i as:

$$OP_i = \{OP_{i,j=\{1..k\}} \mid OP_{ij} = \langle guard_{ij}, weight_{ij}, EVT\_D_{ij} \rangle\}$$

where: i) k is the number of operands in CF Fi, ii) $guard_{ij}$ is the guard of the operand OP_{ij}, iii) $weight_{ij}$ is the weight of the operand OP_{ij}, iv) EVT_D_{ij} are the events that are directly contained in an operand OP_{ij}.

[6] $\rightarrowtail\!\!\!\!\!\rightarrow$ denotes a bijective function
[7] $\twoheadrightarrow$ denotes a total surjection
[8] R^+: the transitive closure of R

We use the following functions to manipulate the operands:

- EVT_D returns the events that are directly contained in each operand[9]:
  $$EVT\_D : OP \rightarrow \mathbb{P}(EVT)$$
- EVT_G returns all the events that are contained in an operand including those which are contained in its nested operands:
  $$EVT\_G : OP \rightarrow \mathbb{P}(EVT)$$
- *weight* returns the weight of each operand:
  $$weight : OP \rightarrow NAT^{+}$$
- $first$ gets the first event of each operand. $first : OP \rightarrow EVT$; intuitively, a first event is an event that has not a preceding events in the considered operand.

$$\textbf{first}=\{(X,e)|X \in OP \ \wedge \ e \in EVT\_G(X) \ \wedge \\ (\forall\ e')[e' \in EVT \ \wedge \ e' <^{*}_{caus} e \Rightarrow e' \notin EVT\_G(X)]\}$$

The instantiation of the definition 2 for SEQ, ALT, OPT and LOOP CF is intuitive and it given in detail in our previous paper [12].
We just present the instantiation of the definition for the PAR CF;

Definition 3 ***(Operands in the PAR combined fragment)***
A parallel combined fragment F_i is composed of a set of k operands:
$$OP_i^{\mathrm{PAR}} = \{OP_{i1}, ..., OP_{ik}\}$$
where $OP_{ij} = \langle True, 1, EVT\_D_{ij} \rangle$
the guard is true and the weight is equal to 1.

The semantics of interactions is explained with an interleaving semantics [1], i.e. two events may not occur at exactly the same time.
In the same way, we choose an interleaving semantics to support alternatives and concurrency behaviours, since it is more appropriate for SD modelling behaviours of distributed system. Indeed, if the semantics allows the occurrence of two events exactly in the same time (like in the true-concurrency semantics [10]), in the case of an ALT CF, we'll have a simultaneous occurrence of the events of different operands, this is not compliant with the standard semantics of this CF where at most one operand among several potential operands must be chosen.

Transformation of SD as a Tree of Operands An SD is encoded as a tree that is composed by a set of linked operands, such that each operand has at maximum one direct ancestor. For instance, the figure 2 illustrates the associated tree for the SD of the Fig.1. A naive way to transform an SD into a tree is to associate a node to each CF or operand. When building the tree of an SD, we always have a root node that represents the complete SD; the

[9] $\mathbb{P}(EVT)$ is the set of subsets E
[10] true-concurrency semantics is a non-interleaving semantics, it supports the occurrence of two events in the same time

process is then breadth-first. Note that the operands of an ALT or a PAR CF are independent, i.e they have disjoint executions. Therefore, to simplify the tree representation of the SD, we substitute the node which should stand for these fragments with the nodes representing their operands. They are moved to the upper level. However, to distinguish them, the operands of the same fragment have their indexes built with the same prefix (OP_{21}, OP_{22} and OP_{23}). From the node of a current SD, the consecutive fragments of the SD become the nodes of the current node. Each fragment is either represented as a node or it is represented by the nodes of its operands. A node is associated to each CF that has only one operand (for instance LOOP or OPT). A CF with more than one operand (for instance ALT or PAR) is replaced with the nodes associated to its operands.

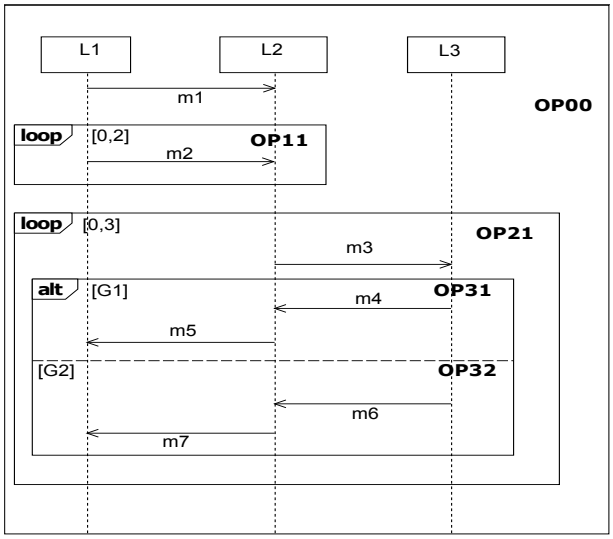

Fig. 1. Example of SD with nested CF

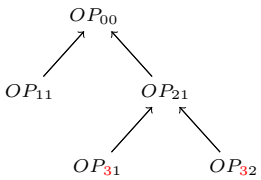

Fig. 2. Tree associated to the SD of the Fig. 1

We define the tree structure for SD operands as follows:

Definition 4 ***(Tree structure for SD operands)***
The tree structure tree_OP related to an SD is defined as a partial function: $tree\_OP : OP \nrightarrow OP$ *which is acyclic and non-reflexive. The root is the only operand that does not have a parent:*
$(\forall X)[X \in OP \ \wedge \ X \notin dom(tree\_OP) \ \wedge \ X \in ran(tree\_OP) \Rightarrow X = OP_{00}]$

Once an SD is transformed to a tree of operands, we define relations that permit to get the locations of the operands that are required in the formalizations of precedence relations. To associate to each operand all the operands where it is nested (its ancestor operands in the *tree_OP*), we introduce the relation *ancestor*. To identify the operands of the same CF ALT, PAR, we introduce the relation *brother*. We call brother operands those that belong to the same CF ALT, PAR. In a given tree: $tree\_OP = \{OP_{i1}....OP_{ij}\}$, the brother operands are the operands that belong to the same level and that have the same index i. Hence, the operands of the same sibling are not all necessarily brothers, since some of them came from lower level when built the tree.

• *ancestor*: a binary transitive relation [11] that is defined on OP.

$$ancestor : OP \leftrightarrow OP$$

For an operand X we compute its ancestors[12] as follows:

$$ancestor[\{X\}] = \bigcup_{s \in \{1,..,d\}} \{tree\_OP^s(X)\}$$

where d is the depth of the node X in the *tree_OP*.

Illustration. In Fig.2, $ancestor[\{OP_{00}\}] = \emptyset$, and $ancestor[\{OP_{31}\}] = \{OP_{21}, OP_{00}\}$.

• *brother*: a binary transitive relation that is defined on a set OP.

$$brother : OP \leftrightarrow OP$$

$$\mathbf{brother} = \{(OP_{ij}, OP_{tk}) | (OP_{ij}, OP_{tk}) \in OP^2 \wedge (i = t \wedge j \neq k))\}$$

Illustration. In Fig.1, the operands OP_{31}, OP_{32} belong to the same CF ALT, thus they are brothers. $brother[\{OP_{11}\}] = \emptyset$ and $brother[\{OP_{31}\}] = \{OP_{32}\}$

Weight of an event The function *weight* was defined on an operand, We overload the function to associate the weight of the path between two operands.

$$weight_e : (OP \times OP) \rightarrow NAT^+$$

For two operands X and Y, we compute the weight of their paths as follows:

$$\left\{ \begin{array}{l} weight_e(X,Y) = 1 \ if \ X = Y \\ weight_e(X,Y) = \prod\limits_{s \in \{0,..,d\}} weight(tree\_OP^s(Y)) \end{array} \right\}$$

[11] $\leftrightarrow$ denotes a relation

[12] $R[\{e\}]$: Relational image; gives the set of images

with d the *length* of the path between the operand X and the operand Y.
We overload the function *weight* that permits to associate to each event its maximal number of occurrence.

$$weight : EVT \rightarrow NAT^{+}$$

For an event evt of an operand X, such that $evt \in EVT\_D(X)$, we compute its weight as follows:

$$\begin{aligned}
&weight(evt) = \\
&weight(X) * weight(tree_OP(X)) * \\
&weight(tree_OP^2(X) * ... * weight(\underbrace{tree_OP^d(X)}_{OP_{00}})) \\
&= \prod_{s \in \{0,d\}} weight(tree_OP^s(X)) \\
&= weight_e(OP_{00}, X), (with\ d = depth\ of\ X)
\end{aligned}$$

The new formalization is used as a basis for the extension of the causal relationships that permits to compute the partial order between the events of the SD.

3.2 Extension of the Causal semantics

The relations $<_{Sync}$, $<_{RE}$, $<_{EE}$ and $<_{RR}$ permit to compute the precedence relations for each event of an SD. The structuring of SD with nested CF in form of tree permits an obvious identification of the preceding events, they are grouped by operand, for each event that belongs to this kind of SD.
In this section, we generalize these relations. The synchronisation relationship ($<_{Sync}$) is unchangeable. The formalizations of $<_{RE}$ and $<_{EE}$ relationships permit to order two events that belong to the same lifeline and that are successive. We define a new relationship $<_{RR}$ to consider some particular cases of the ordering of receiving of events in the context of distributed components.
To detail a bit, and to alleviate the presentation of the formalization of $<_{RE}$ and $<_{EE}$ relationships, we introduce three binary relations *not_in_brother*, *succ*1 and *succ*2. In the following, we first give the intuition of each of them before their formalizations.
Two successive events that belong to distinct operands of an ALT or a PARCF must not be ordered. The relation *not_in_brother* expresses this intuition: the successive events of an ALT CF to be ordered must neither belong to brother operands nor to operands where in their respective ancestors exist a brother operands.

$$\begin{aligned}
&\textbf{not_in_brother}=\{(e,e') | (e,e') \in EVT^2 \wedge (\forall X)(\forall Y) \\
&[X \in (ancestor[\{EVT_D^{-1}(e)\}] \cup \{EVT_D^{-1}(e)\}) \\
&\wedge Y \in (ancestor[\{EVT_D^{-1}(e')\}] \cup \{EVT_D^{-1}(e')\}) \\
&\Rightarrow (X,Y) \notin brother]\}
\end{aligned}$$

Illustration: in Fig.1, the event $!m4 \in OP31$, the event $!m6 \in OP32$, however we have $OP32 \in brother[\{OP31\}]$, hence the events $!m4$ and $!m6$

should not be ordered.
Formally, we define that two events are successive in two manners with two distinct relations *succ*1 and *succ*2. These relations are used respectively in the formalization of $<_{EE}$ and $<_{RE}$ relationships. The relation *succ*1 relates two events that belong to the same lifeline and which are successive. Nevertheless, we admit between them, events that must necessarily belong to an operand that can be omitted (i.e. the events between successive events do not belong to any operand ancestor of the operands of the considered events).

$$\begin{aligned}&\textbf{succ1}=\{(e,e')|(e,e')\in EVT^2\wedge\\&(\exists l)[l\in L\ \wedge\ e<^*_{SD,l}e'\\&\ \wedge(\forall e")[e"\in EVT\ \wedge\ (e<^*_{SD,l}e"\ \wedge e"<^*_{SD,l}e')\\&\Rightarrow EVT\_D^{-1}(e")\notin(ancestor[\{EVT\_D^{-1}(e)\}]\\&\cup ancestor[\{EVT\_D^{-1}(e')\}])]]\}\end{aligned}$$

The relation *succ*2 expresses the same conditions and effects as those defined in *succ*1 relationships, moreover it expresses that we admit between the successive events received events.

$$\begin{aligned}&\textbf{succ2}=\{(e,e')|(e,e')\in EVT^2\wedge\\&(\exists l)[l\in L\ \wedge\ e<^*_{SD,l}e'\ \wedge\ (\forall e")[e"\in EVT\ \wedge\\&(e<^*_{SD,l}e"\ \wedge e"<^*_{SD,l}e')\\&\Rightarrow e"\in ran(FCT\_r)\vee\\&EVT\_D^{-1}(e")\notin(ancestor[\{EVT\_D^{-1}(e)\}]\\&\cup ancestor[\{EVT\_D^{-1}(e')\}])]]\}\end{aligned}$$

The relationship $<_{EE}$ permits to order two sent events that satisfy the conditions expressed in *not_in_brother* and *succ*1 relations.

$$\begin{aligned}&<_{EE}=\{(e,e')|[(e,e')\in(EVT)^2\wedge\\&e\in ran(FCT\_s)\ \wedge e'\in ran(FCT\_s)\wedge\\&(e,e')\in not\_in\_brother\ \wedge(e,e')\in succ1]\}\end{aligned}$$

The relationship $<_{RE}$ permits to order two events such that the first one is a received event and the second one is a sent event, and both of them satisfy the conditions expressed in *not_in_brother* and *succ*2 relations.

$$\begin{aligned}&<_{RE}=\{(e,e')|[(e,e')\in(EVT)^2\wedge\\&e\in ran(FCT\_r)\ \wedge e'\in ran(FCT\_s)\wedge\\&(e,e')\in not\_in\_brother\ \wedge(e,e')\in succ2]\}\end{aligned}$$

In a distributed system context, the components are independent and the communication between them is carried out according to protocols, each of them guarantees properties semantics concerning the reception of messages. In case the considered protocol ensures a First in First Out (FIFO) delivery order, the receptions of two messages coming from the same lifeline are received in the same order of their emission. The $<_{RR}$ relationship permits

to compute these precedence relations.

$$
\begin{aligned}
&<_{RR}=\{(e,e') \mid [(e,e') \in E_r^2 \ \wedge \\
&(\exists e_1, \exists e_2)[(e_1,e_2) \in E_s^2 \ \wedge \\
&\ Fct\_s^{-1}(e_1) = Fct\_r^{-1}(e) \ \wedge \\
&\ Fct\_s^{-1}(e_2) = Fct\_r^{-1}(e) \ \wedge \\
&\ e_1 <_{EE}^{*} e_2 \ \wedge \ Fct\_l(e_1) = Fct\_l(e_2)]\}
\end{aligned}
$$

In the previous work [12], we showed that in LOOP CF as well as in nested CF that contains LOOP CF the determination of the precedence relations for each event is not obvious.

3.3 Hidden Precedence Relations in LOOP Combined Fragment

The events inside a LOOP operand can have as preceding events that can be located:

- for the first iteration: *i)* either outside the LOOP operand and/or, *ii)* inside the LOOP operand of the same iteration.
- from the second iteration: *i)* either outside the LOOP operand and/or, *ii)* inside the LOOP operand of the same iteration and/or of the previous iterations.

We call hidden relations the relations between the events of LOOP operand of the current iteration and the events of the previous iterations (Fig.**??**). These relations appear when the LOOP operand is flatten at least one time. Hence, the necessity of defining a new relation $<_{Hcaus}$ in which we express the constraints of precedence between the events of the current iteration and the events of the previous iteration. In order to compute the hidden precedence relations, we propose the following steps: we flatten the LOOP operand only once whatever is the number of iterations; we obtain an intermediate sequence diagram SD'.
In SD', we rename the operands as well as the events of the second iteration with the same name as those of the preceding iteration by labelling them with a single quote (Fig. 3). We define the set EVT' to represent the events of the next iteration. $<'_{RE}$ and $<'_{EE}$ are respectively the reception-emission, and the emission emission relationships associated to the SD'. In an SD we can have several LOOP operand that can be sequenced or nested. In this case, the same processing is applied by computing for each LOOP operand its hidden relationships; we note $<_{HcausX}$, the hidden relations of a given LOOP operand named X. The formalization of the hidden relationships for a LOOP operand X is given as follows.

$$
\begin{aligned}
&<_{HcausX}= \\
&\{(e,e') | e \in EVT \wedge e' \in EVT' \wedge \\
&(e,e') \in <'_{RE} \vee (e,e') \in <'_{EE}\}
\end{aligned}
$$

Illustration1. Consider the SD in Fig.**??**, the SD' represents the flattening of the LOOP operand only once. In the SD', in the first iteration, the $!m2$ has as

preceding event the event $!m1$ that is located outside the loop operand; the event $!m3$ has as preceding events, the event $?m1$ (that is located outside the loop operand) and the $?m2$ (that belongs to the same iteration). In the second iteration, the event $!m2'$ has as preceding event the event $!m4$ which belongs to the first iteration; the event $!m3'$ has as preceding events the events $?m4$ and $?m2'$, which belong respectively to the first and the second iteration.

Illustration2. As aforementioned, for an ALT CF, only one operand must be executed, hence the events that belong to distinct operands must not be ordered, otherwise we'll have deadlocks of some events.

However, in some particular cases of nested structure, especially for an ALT that is nested in a LOOP CF, we can face a problem that the events of distinct operands of the same ALT CF (brother operands) can have precedence relations. Figure 6 represents a possible execution of the SD (depicted in Fig.4) containing nested CF. In the first iteration of the LOOP CF, the first operand of the ALT CF is executed; in the second iteration of the LOOP CF, the third operand of the ALT CF is executed. According to the $<_{EE}$ relationship, the event $!m2$ precedes the event $!m7'$, although they respectively belong to brother operands $OP21$ and $OP22$. Likewise for the events $!m3$ and $!m6'$. This is problematic, since the events of brother operands should not be ordered. This justifies the renaming of the events and the operands of the next iteration to avoid this issue.

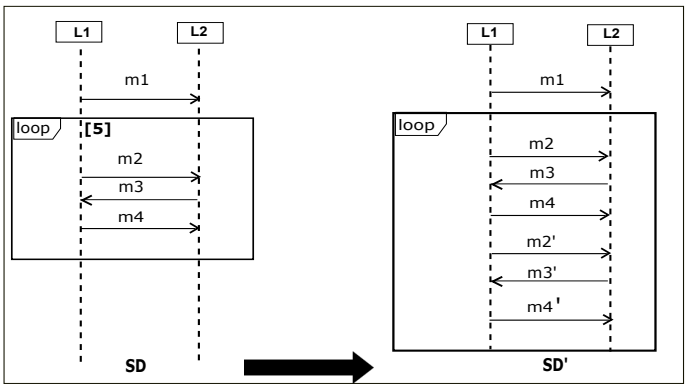

Fig. 3. Processing of an SD with LOOP operand

In an SD we can have several LOOP operand that can be sequenced or nested. In this case, the same processing is applied by computing for each LOOP operand its hidden relationships; the entire hidden relation is the union of the hidden relations of each LOOP operand. Now, the causal relationships is computed as follows.

$$<_{caus}=<_{SYNC} \cup <_{RE} \cup <_{EE} \cup <_{RR} \cup <_{Hcaus}$$

That means the ordering of events depends on the cumulative rules of the relationships. The valid traces are those which can be generated satisfying these

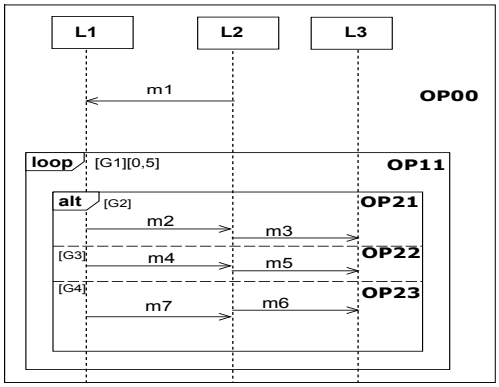

Fig. 4. SD with nested CF

orders.
The defined rules ($<_{RE}$, $<_{EE}$, $<_{RE}$ and $<_{Hcaus}$) may be applied to the standard semantics by restoring the constraints that we relaxed. In the same way, these rules can be adapted for any kind of semantics by strengthening or weakening some constraints. The causal semantics can be exploited for several purposes, it can used as basis for the computation of all possible valid traces of SD modelling behaviours of distributed systems as it can be the basis for the definition of an operational semantics that facilitates its implementation and then the analysis of the SD and several properties of systems for instance safety, liveness, fairness or reachability properties.

4 Operational Semantics

The most of the existing semantics are trace-based semantics, they require a meticulous work that consists in generating all possible traces of an SD then in their categorisation depending on the aim of the semantic, and they do not propose tools to ensure this task [3] [16]. Moreover, most of them ignore interaction constraint that guards combined fragments, which are essential to ensure soundness of refinement relation [16]. In the approach of [16], the authors consider the interaction constraint in a non-intuitive way. Indeed, they propose to include the guard as an element in the standard definition of trace.

The motivation behind the definition of an operational semantics is the intention of the use of existing refinement relations that are well defined on transition systems, since an operational semantics is concretely given as a transition system. Moreover, in the operational semantics, we define execution strategies of the events of an SD with nested CF. They include on the one hand, the order of the occurrence of the events in a nested structure (CF) as well as the conditions under which those executions can take place, on the other hand their execution effects that they produce. These strategies allow for better understanding and analysis of the behaviour of a sequence diagram.

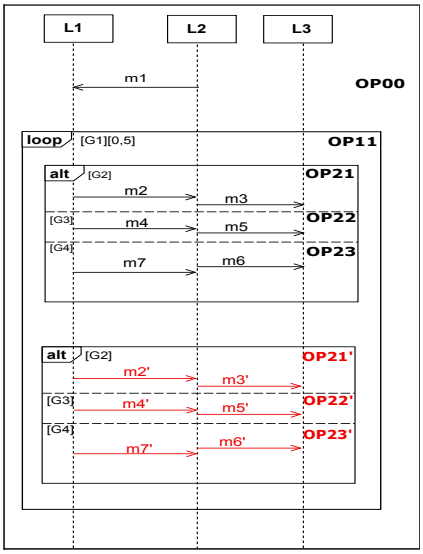

Fig. 5. Processing of the OP11 LOOP operand of Fig.4

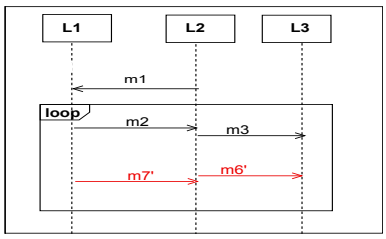

Fig. 6. Possible execution of the SD of Fig.4

Moreover, the guard is straightforwardly expressed. Formally, it is given as a guarded transition system:

$$Sem(SD) = \langle S, S^0, \triangle \rangle$$

where S is the set of possible states of the SD, S^0 is the initial state and $\triangle$ is the transition relation.

4.1 State

Each state of an SD is expressed with two variables (*state*, *current_instance*): *state* expresses the states of all events of SD, *current_instance* expresses the lifeline of the current event.

The state of an event. An event which belongs to a basic SD can have two obvious basic states: executed or not yet executed. In our semantics, we support sequence diagrams with sequential CF that can be nested. The basic states are not sufficient to express the state of an event in an SD with sophisticated structures (nested CF). Indeed, each event in such SD can be: not yet occurred, occurred, consumed one or several times. Then, the variable *state* is defined as follows.

$$state : EVT \rightarrow NAT$$

The state of an event is decreased whenever it is occurred or ignored. To describe the state of an event e, we use the following vocabulary:

- not yet occurred: when $state(e) = weight(e)$,
- occurred: if the event e is executed or ignored one or several times and $0 < state(e) < weight(e)$,
- consumed: when $state(e) = 0$.

During its execution, an SD can be in one state among the following states:

- an initial state S^0, when all its events are not yet occurred,
- an intermediate state of S,
- a final state, when all its events are consumed: $state = EVT \times \{0\}$.

The notion of state is very important, indeed, it constraints the occurrence of a given event (for instance we decrement the state of an event whenever it is occurred, or if we want to prohibit its occurrence); it also serves to indicate the location of the considered event; this information is useful especially when we have several nested LOOP CF.

4.2 Transition Rules

For each event evt in an SD we associate the following transition:

$$p \xrightarrow{[g]evt} q \stackrel{def}{\equiv} ((p,\ [g]\ evt,\ q) \in \triangle\ \wedge\ g)$$

An event is enabled only when its trigger conditions, (labelled **TCi**), hold. When the enabled event occurs, it produces execution effects (labelled **EEi**) that update the SD from the state p to the state q.

In the following, we define rules for the guarded transition system which constraint the occurrence of the events (the trigger conditions and the executions effects). The rules of our operational semantics have the following shape.

$$evt = \frac{CD1 \wedge CD2 \wedge ...CDi}{EE1, EE2, ..., EEi}$$

4.3 Occurrence of the events

For each event the trigger conditions must be checked conjointly and the executions effects are produced simultaneously.

Trigger conditions Some trigger conditions have a simple shape : they are atomic formulas where others trigger conditions are composed by the conjunction of several conditions. Indeed, some conditions must be strengthened in order to take into account of some particular cases and to prevent some issues that result of the presence, the disposition of the nesting of some CF (for instance the nesting CF that contains LOOP CF that induces hidden relations).

• First trigger condition related to satisfaction of precedence constraints

In our causal semantics, we first transform the considered SD in the form of a tree of operands. This transformation allows us to identify easily the preceding events of each event that are grouped by operand. Then the defined relations permit the computation of the precedence relations between the events.
The first trigger condition **TC1** necessary to the occurrence of each event consists in checking that its preceding events were occurred. This is made by comparing the states of the considered event and those of its preceding events. Remind that each event has a state which is initialized to its weight corresponding to its maximal number of occurrence. Depending on the kind of combination of CF ((ALT-LOOP), (ALT -LOOP), (LOOP-LOOP)....) to nest and the location of the considered events (these informations is given in the states of the events), the shape of the first trigger condition varies.

Consider an event *evt* that belongs to an UML2.X SD. To facilitate the reasoning, we assume that the event *evt* has only one preceding event *e*. The occurrence of the event *evt* depends on the state of the event *e*.

If the considered events *e* and *evt* have the same weight, then the trigger condition is simply expressed in the form of an inequality on the respective states of *evt* and *e*, hence it is enough to to check that:

$$state(e) < state(evt)$$

However, in an SD, we can have several combinations of different kinds of CF. The combinations and the nesting of some kinds CF, especially those that contain LOOP CF complicate the form of the first trigger condition. Indeed if the events have distinct weights that are > 1, it is the case where the events belong to nested CF that contain loop CF. The weight is a term making the product from the root to the event. The weight of an intermediate operand is a multiplicative factor of the events contained in the child operands.
Therefore, the comparison of the states of two events is based on their weights relative to a common node (operand) or the first shared node that encompasses the events, which is the lowest common ancestor (LCA). Indeed, the terms of the weight derived from the ancestors are the multiplicative factors common.
For instance in the figure 13, consider the events $?m1$ and $!m2$ that belong respectively to $OP21$ and $OP11$ operands, the LCA is the operand $OP11$, hence $weight(?m1) = 3 * 5$ and $weight(!m2) = 5$. In the figure 17, consider the events $!m1$ and $!m2$ that belong respectively to $OP21$ and $OP31$ operands, the LCA is the operand $OP11$, hence $weight(!m1) = 5 * 3$ and $weight(!m2) = 5 * 4$.

Consider the operands X et Y of the events e and evt : $X = EVT\_D^{-1}(e)$, $Y = EVT\_D^{-1}(evt)$ and Z is the lowest common ancestor of the operands X and Y: $Z = LCA(X, Y)$. Depending on the weights of the events of e and evt, we distinguish the following cases:

1. **Case1**: each of the event e and evt has a weight that is equal to 1. In this case None of the operands X Y or Z is a LOOP operand. Moreover their respective ancestors are not LOOP operands.

2. **Case2**: each of the events e et evt has a weight that is different of 1. In this case, we have to argue with regard to the lowest common ancestor (Z) of the operands X and Y of the events e and evt. Indeed, we distinguish 4 possible cases:
   - 2.1 **Case 2.1**: There is no LOOP operand neither in the path from the operand Z to the operand X nor in the path from the operand Z to the operand Y (i.e. $weight(Z, X) = 1$ and $weight(Z, Y) = 1$),
   - 2.2 **Case 2.2**: there is a LOOP operand only in the path from the operand Z to the operand X (i.e. $weight(Z, X) > 1$ and $weight(Z, Y) = 1$),
   - 2.3 **Case2.3**: there is a LOOP operand only in the path from the operand Z to the operand Y (i.e. $weight(Z, X) = 1$ et $weight(Z, Y) > 1$),
   - 2.4 **Case2.4**: in each path from the operand Z to the operand X and from the operand Z to the operand Y there is a LOOP operand (i.e. $weight(Z, X) > 1$ et $weight(Z, Y) > 1$).

In the sequel, we illustrate each case with an example et we give the appropriate trigger condition.

• **Case1.** The weight of event e and evt is equal to 1. We distinguish two possible cases: *i)* both events e and evt are located in the same operand: $X = EVT\_D^{-1}(e) = EVT\_D^{-1}(evt)$ (see Fig.7), and *ii)* the events are located in distinct operands: $EVT\_D^{-1}(e) \neq EVT\_D^{-1}(evt)$ (see Fig.8).
In this case it is enough to check that :

$$CD11 : state(e) = 0 < state(evt) = 1$$

Illustration: in both Figure 7 and Figure 8, according to the $< EE$ relation, the events $!m1$ and $!m2$ are ordered, they are respectively located in the same operand (Fig.7) and in distinct operands 8. Both events have a weight equal to 1. The event $!m2$ can occur only if the event $!m1$ was consumed. Hence we must check the condition

$$state(!m1) = 0 \ \wedge \ state(!m2) = 1$$

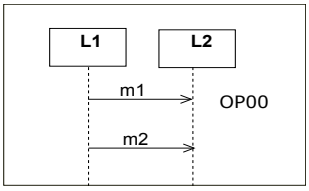

Fig. 7. SD0: $!m1 < !m2$

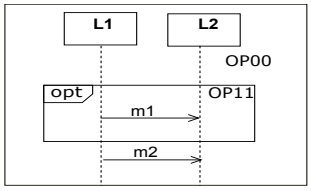

Fig. 8. SD1: $!m1 < !m2$

Fig. 9. Illustration case1

• **Case2.1**. The weight of the events e and evt are different of 1. We distinguish two cases: *i)* the events are located in the same operand:

$X = EVT\_D^{-1}(e) = EVT\_D^{-1}(evt)$ (see Fig.10), and *ii)* the events evt and e are located in distinct operands: $EVT\_D^{-1}(e) \neq EVT\_D^{-1}(evt)$ (Fig. 11). We consider only the case where the paths from the operand Z to the operand X and from the operand Z to the operand Y did not contain a LOOP operand (i.e. $weight(Z, X) = 1$ and $weight(Z, Y) = 1$). In this case, either the operand Z or at least one of its ancestors is a LOOP operand.
Illustration1: in the figure 10, the weight of each event $!m1$ and $?m1$ is equal to 4, hence each of them can occur 4 times. For each iteration, the message $m1$ can be received only if it is sent (the event $!m1$ was occurred) This conditions constraints the occurrence of the event $?m1$, it is expressed as follows:

$$state(!m1) < state(?m1)$$

Illustration2: in the figure 11, the weight of each event $!m1$ and $?m1$ is equal to 4, hence each of them can occur 4 times. For each iteration, the event $?m2$ can occur only if the event $!m1$ was occurred. This condition constraints the occurrence of the event $?m1$, it is expressed as follows:

$$state!m1) < state(!m2)$$

Hence in these cases the trigger condition can be expressed as follows:

$$CD12 : \ state(e) < state(evt)$$

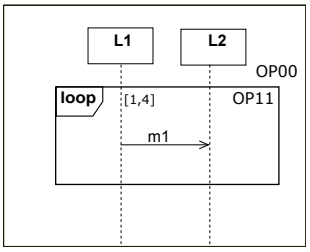

Fig. 10. SD2: $!m1 <?m1$

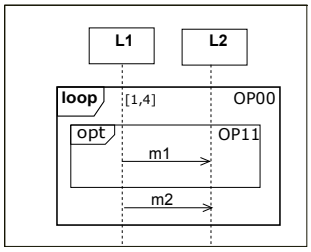

Fig. 11. SD3: $!m1 <!m2$

Fig. 12. Illustration case2.1

• **Case2.2**. The weights of the events e and evt are different of 1. Moreover, they are located in distinct operands: $EVT\_D^{-1}(e) \neq EVT\_D^{-1}(evt)$. We have only a LOOP operand in the path from the operand Z to the operand X (i.e. $weight(Z, X) > 1$ and $weight(Z, Y) = 1$)
Illustration: consider the figure 13, for each iteration of the operand $OP11$, the event $!m2$ can occur only if the event $?m1$ was occurred 3 times. The table represented in Fig. 14 illustrates the 5 states for which the event $!m2$ can occur.

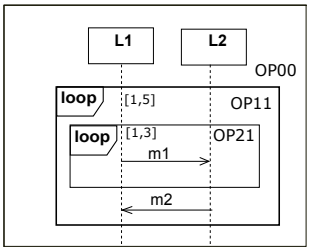

Fig. 13. SD5: $?m1 <!m2$

n° iteration	$state(?m1)$	$state(!m2)$
1	12	5
2	9	4
3	6	3
4	3	2
5	0	1

Fig. 14. Variation of states values with iteration of the SD of Fig.13

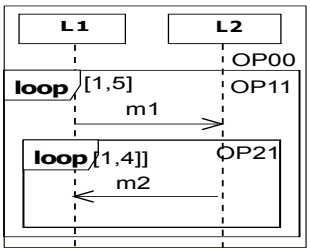

Fig. 15. SD8: $?m1 <!m2$

n° iteration	$state(?m1)$	$state(!m2)$
1	4	20..17
2	3	16 ..13
3	2	12 ..9
4	1	8..5
5	0	4 ..1

Fig. 16. Variation of states values with iteration of the SD of Fig.15

$$[state(?m1)/3 < state(!m2)] \ \wedge \ [(state(?m1) mod\ 3 = 0)]$$

In this case, the trigger condition of the event evt is expressed in form of a conjunction of predicates. Such that the first predicate is an inequality on states of the event evt and its preceding event e where the state of the preceding event is weighted with the coefficient $1/weight(Z, X)$. The second predicate permits to the event evt iterate once the event e was occurred $weight(Z, X)$ times.

$$CD13 : [state(e)/weight(Z, X) < state(evt)] \wedge [(state(e) \bmod weight(Z, X) = 0)]$$

• **Case2.3**. The weights of the events e and evt are different of 1. Moreover, they are located in distinct operands: $EVT\_D^{-1}(e) \neq EVT\_D^{-1}(evt)$.
We have only a LOOP operand in the path from the operand Z to the operand Y (i.e. $weight(Z, X) = 1$ et $weight(Z, Y) > 1$).
Illustration: in the figure 15, the event $?m1$ precedes the event $!m2$. For each iteration of the operand $OP11$, the event $!m2$ occurs 4 times. For each occurrence of the event $?m1$, the event $!m2$ occurs 4 times. The table represented in Fig. 16 illustrates the 20 states for which the event $!m2$ can occur. Hence the trigger condition of the event $!m2$ can be expressed as follows.
$(state(!m2) \bmod 4 = 0) \Longrightarrow (state(?m1) < state(!m2)/4)$
Hence, in this case, the trigger condition of the event evt is expressed as follows.
$CD14 : (state(evt) \bmod weight(Z, Y) = 0) \Longrightarrow (state(e) < state(evt)/weight(Z, Y))$
• **Case2.4**. The weights of the events e and evt are different of 1. Moreover, they are located in distinct operands: $EVT\_D^{-1}(e) \neq EVT\_D^{-1}(evt)$.

In each path (from the operand Z to the operand X and from the operand Z to the operand Y), it exists a LOOP operand (i.e. $weight(Z, X) > 1$ et $weight(Z, Y) > 1$).
Illustration: in the figure 17, the event $!m1$ precedes the event $!m2$. For each execution of the operand $OP11$ the event $!m2$ occurs 4 times. After each 3 occurrences of the event $!m1$, the event $!m2$ occurs 4 times. The table represented in Fig. 18 illustrates the 20 states where the event $!m2$ can occur.

$state(!m2)$ mod $4 = 0 \Rightarrow state(!m1)/3 < state(!m2)/4 \wedge (state(!m1)$ mod $3 = 0)$

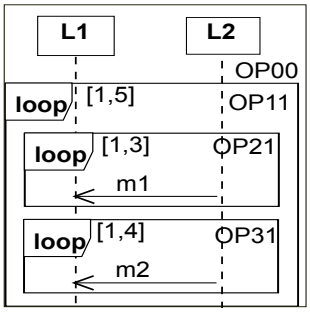

Fig. 17. SD7: $!m1 < !m2$

n° iteration	$state(!m1)$	$state(!m2)$
1	12	20..17
2	9	16 ..13
3	6	12 ..9
4	3	8..5
5	0	4 ..1

Fig. 18. Variation of states values with iteration of the SD of Fig.17

In this case, the trigger condition of the event evt is expressed as follows.

$$CD15: \ (state(evt) \text{ mod } weight(Z, Y) = 0) \Rightarrow [state(e)/weight(Z, X) < state(evt)/weight(Z, Y)] \wedge [(state(e) \text{ mod } weight(Z, X) = 0)]$$

Generalization of the trigger condition $TC1$. Thenceforth, we can deduce the general form of the first trigger condition. We check the occurrence of the preceding events that are computed receptively from the relation ($<_{caus} \setminus <_{Hcaus}$) and from the relation $<_{Hcaus}$ independently in two distinct trigger conditions.

- $(e, evt) \in (<_{caus} \setminus <_{Hcaus})$ $(TC1)$
- $(e, evt) \in <_{Hcaus}$ $(TC1')$

• Hence, when we have:
$\forall(e)(\exists X)(\exists Z)[(e, evt) \in (<_{caus} \setminus <_{Hcaus}) \ \wedge \ X = EVTD^{-1}(e)$
$\wedge \ Y = EVTD^{-1}(evt) \ \wedge \ Z = LCA(X, Y)$

Then, the first trigger condition is expressed as follows:

TC1:
$(state(evt)$ **mod** $weight(Z,Y) = 0) \Longrightarrow$
$((state(e)/weight(Z,X) < state(evt)/weight(Z,Y)$
$\wedge\ (state(e)$ **mod** $weight(Z,X) = 0)))$

• For each event *evt* of a LOOP operand or that belong to a nested CF that contains a LOOP operand and that has hidden preceding events that appear from the second iteration.
Hence, when we have:
$(\forall e)(\exists X)(\exists Z)[(e, evt) \in <_{Hcaus} \ \wedge\ X = EVTD^{-1}(e)$
$\wedge\ Y = EVTD^{-1}(evt)\ \wedge\ Z = LCA(X,Y)$
We define the following trigger condition **TC1'**.

TC1':
$(state(e)$ **mod** $(weight(Z,Y) * weight(Z) <> 0)) \Longrightarrow$
$((state(e)/weight(Z,X) = state(evt)/weight(Z,Y)$
$\wedge\ (e, evt) \in not\_in\_brother))$

• the second trigger condition consists in checking that the event can still be occurred: it is not yet consumed (**TC2**). It is formally defined as follows:

$$\textbf{TC2}:\ state(evt) \geq 1$$

• for the events that belong to a guarded CF we add a third trigger condition that permits to check the value of the guard.

Execution effects . The execution effects of an event should simultaneously:

- update the state of the current event by decreasing its state (**EE1**);
  the execution effect **EE1** is to update the state with: **EE1**: $state(evt) - 1$
- update the lifeline of the current event (**EE2**); the execution effect **EE2** is to set *current_instance* with: **EE2**: $current\_instance := FCT\_l(evt)$ Remind the $FCT\_l(evt)$ gives the lifeline of the event.

Particular cases: for the guarded ALT, we assume that the evaluation of the guard is made on the first event. If the guard is evaluated to true (**TC3**) then the first event must synchronize the events of the other operands of the same CF by decrementing their states (remind that the standard semantics of the ALT CF impose that only one operand must be executed among several potential operands having simultaneously a true guard). Otherwise, if the guard is

evaluated to false (**TC3'**), the first event must decrement the states of the events of the same operand in order to prohibit their occurrence. Hence, in addition to the trigger conditions **TC1** and/or **TC1'**, we must add a third trigger condition **TC3** for the first event of each operand of an ALT CF.

$$\textbf{TC3} : guard := true$$
$$\textbf{TC3'} : guard := false$$

If the guard is evaluated to true, in addition to the executions effects **EE1** and **EE2** , we must add a third execution effect **EE3** to modify the states of the events of the brother operands. When we have $[evt \in EVT\_D(X) \ \wedge \ e \in EVT\_D(Y) \ \wedge \ e \in EVT\_G(Z)]$,
EE3: $state(e) - weight(Z, Y)$, where $Z = brother(X)$ If the guard is evaluated to false, in addition to the executions effects **EE1** and **EE2** , we must add a third execution effect **EE3'** to decrement the states of the events of the same operand. When we have $evt \in EVT\_D(X) \ \wedge \ e \in EVT\_D(Y) \ \wedge \ e \in EVT\_G(X)$, then
EE3': $state(e) - weight(X, Y)$
All the operands of any kind of CF can be guarded, in this case in addition to the trigger conditions **TC1** and/or **TC1'**, we must add a third trigger condition **TC3** for the first event of the considered operand. If the guard is evaluated to false, in addition to the executions effects **EE1** and **EE2** , we must add a third execution effect **EE3'** to decrement the states of the events of the same operand. N.B In a nested CF, we assume that the guard evaluation of a child operand should be made after a $True$ guard evaluation of the parent operand. This is compliant with the hypothesis we made in Subsection 3.1, which states that each operand has one first event. Moreover, if the guard of the parent operand is evaluated to $False$, its events including the events of its child operands are ignored.

All these rules define the operational semantics of UML2.X SD with nested combined fragments. They are not linked to any target formalism and they can be implemented in various ways and by any formalism doted with tools for its checking.

5 Related works

In the literature, there are several semantics approaches to define a semantics for UML2.X SD. Among them we cite the most popular: *i)* denotational semantics [3], [16], *ii)* transformational semantics [9], and *iii)* operational semantics [17], [10]. They are mainly proposed to overcome some issues of the standard semantics, or to adapt the use of the SD to the modelling of different systems, and for other purposes. For instance, in [3] and [16], the authors defined a trace semantics based on denotational semantics to distinguish between mandatory and required behaviours. In [10], the authors proposed a denotational semantics based on partially ordered multisets or pomsets that deals with language constructs for specifying negative traces. In the works of [9], the authors proposed a

transformational semantics based on the translation of SD into Büchi automata in order to verify liveness and safety properties of reactive systems. In [17], the authors proposed an operational semantics for SD that supports negative behaviours and that distinguishes between possible and required behaviours. In [10], the authors proposed an operational semantics for SD, which is compliant with the semantics proposed in [18], for capturing the composition operators from High Message Sequence Charts (HMSC) and NEG ASSERT CF.

We underlined that a few of the existing semantics, [16], [3], [9], [18], can be used to formalize the refinement relation while the others do not allow it [19], [4], [20], [17], [2].

For this purpose, the trace-based semantics, [16], [3], [9], are not very convenient, indeed they permit to verify only some kinds of refinement relations (trace inclusion, trace equivalence...), this require a meticulous preprocessing on all traces of the considered SD, knowing that most of them did not propose tools that ease this arduous task; moreover they ignore the guards of CF which are essential to ensure soundness of refinement relation. Although, in the work of [16], the proposed trace-based semantic refinement considers guard, but in a non-intuitive way, by modifying the standard definition of the trace.

In contrast to trace-based semantics, with an operational semantics several kinds of well-defined refinement relations can be expressed (simulation trace, inclusion trace, equivalence trace...). Moreover the operational facilitates the analysis of the behaviours of the modelled systems.

Most of the existing semantics [16], [3], [9], [18], [4], [17] are usually based on the definitions of the standard for the computation of traces, thus they are not suitable for SD modelling behaviours of distributed systems. Moreover most of the work, [5], [6], [21], [22] did not deal properly with some CF and the nested CF. Indeed they impose strict hypothesis to avoid inconsistencies due to the use of these CF. In our last work we have well explained these restrictions that limit the expressive power of these CF. To overcome these insufficiencies, we proposed an operational semantics that is, on the one hand, based on an extended causal semantics, suitable for UML2.X SD equipped with the most popular CF modelling distributed systems, on the other hand, it supports guards straightforwardly since it is given as a guarded transition system. The operational semantics can be easily implemented and can be used as a basis for refinement checking purpose for our ongoing work.

6 Conclusion

To help in preliminaries design steps of distributed systems, we have equipped UML2.X sequence diagrams with a causal semantics that is based on partial order theory and tree structure. Its relations permit the determination of the precedence relations straightforwardly for SD with nested CF that model behaviours of a distributed system, by avoiding its flattening, hence the compact syntactic representation is preserved. The causal semantics can serves for several

purposes, in this paper we have proposed an operational semantics in which we define execution strategies of the events of an SD with nested CF. The proposed operational semantics is not linked to a specific target formalism. We currently implement the operational semantics with the Event-B method [11], [23]. Transforming SD into corresponding B specifications enables rigorous model analysis using the formal techniques of Event B and its various tools Rodin: (with a theorem-prover, and with ProB model-checker). Meanwhile, the operational semantics serves as the basis of our ongoing work on the verification of the refinement relation between sequence diagrams. Indeed the operational semantics is concretely given as a transition system since refinement relations are well defined on the transition system as a simulation relation. This is used for investigating whether or not a sequence diagram specification is a correct refinement of another sequence diagram specification. In addition, we currently study theoretical properties that are derived from the proposed semantics.

References

1. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.2, 2015.
2. Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In Thomas Khne, editor, *Models in Software Engineering*, pages 42–51. Springer, 2006.
3. Ragnhild Kobro Runde øystein Haugen, Knut Eilif Husa and STAIRS. Towards Formal Design with Sequence Diagrams. In *Software and System Modeling*, volume 4, pages 355–357. John Wiley & Sons, Inc., 2005.
4. Harald Störrle. Semantics of Interactions in UML 2.0. In *HCC*, pages 129–136, 2003.
5. Youcef Hammal. Branching Time Semantics for UML 2.0 Sequence Diagrams. *Lecture Notes in Computer Science: Formal Techniques for Networked and Distributed Systems - FORTE 2006*, pages 259–274, 2006.
6. Zoltàn Égel Andràs Kvi Zoltàn Micskei Gàbor Huszerl, Hélène Waeselynck (ed.). Refined design and testing framework, methodology and application results. 2008.
7. Hui Shen. *A Formal Framework for Analyzing Sequence Diagram.* PhD thesis, 2013.
8. David Harel and Shahar Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and System Modeling*, 7(2):237–252, 2008.
9. R. Grosu and SA Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *5th Int. Conf. on Application of Concurrency to System Design*, page 614, 2005.
10. Mara Victoria Cengarle, Peter Graubmann, Stefan Wagner, and Technische Universität München. Semantics of UML 2.0 Interactions with Variabilities, 2005.
11. Fatma Dhaou, Inès Mouakher, Christian Attiogbé, and Khaled Bsaies. Extending Causal Semantics of UML2.0 Sequence Diagram for Distributed Systems. *ICSOFT-EA 2015 - Proceedings of the 10th International Conference on Software Engineering and Applications, Colmar, Alsace, France*, pages 339–347, 2015.
12. Fatma Dhaou, Inès Mouakher, Christian Attiogbé, and Khaled Bsaïes. A Causal Semantics for UML2.0 Sequence Diagrams with Nested Combined Fragments. In *ENASE 2017 - Proceedings of the 12th International Conference on Evaluation*

of Novel Approaches to Software Engineering, Porto, Portugal April 28-29, 2017., pages 47–56, 2017.
13. C.Sibertin-Blanc O.Tahir and J.Cardoso. A Causality-Based Semantics for UML Sequence Diagrams. In *23rd IASTED International Conference on Software Engineering*, pages 106–111. Acta Press, 2005.
14. Tahir O. Sibertin-Blanc, C. and Cardoso J. Interpretation of UML Sequence Diagrams as Causality Flows. In *Advanced Distributed Systems, 5th Int. School and Symposium (ISSAD)*, number 3563, pages 126–140. Acta Press, 2005.
15. J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.
16. Dae-Kyoo Kim. Lunjin Lu. Required Behavior of Sequence Diagrams: Semantics and Refinement. *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 127–136, 2011.
17. M. S. Lund and K. Stlen. A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. *Lecture Notes in Computer Science*, (4085):380395, 2006.
18. Maria Victoria Cengarle and Knapp Alexander. UML 2.0 Interactions: Semantics and Refinement. *Technische Universitat Munchen*, pages 85–99, 2004.
19. Demissie B. Aredo. A Framework for Semantics of UML Sequence Diagrams. *in PVS Journal of Universal Computer Science (JUCS)*, pages 674–697, July 2002.
20. Seung Mo Cho, Hyung Ho Kim, Sung Deok Cha, and Doo Hwan Bae. A semantics of sequence diagrams. *Information Processing Letters*, 84(3):125 – 130, 2002.
21. Alessandra Cavarra and Juliana Küster Filipe. Formalizing Liveness-Enriched Sequence Diagrams Using ASMs. In *Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop, Lutherstadt Wittenberg, Germany, Proceedings*, pages 62–77, 2004.
22. Shahar Maoz, David Harel, and Asaf Kleinbort. A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 2011.
23. Fatma Dhaou, Inès Mouakher, Christian Attiogbé, and Khaled Bsaïes. Refinement of UML2.0 Sequence Diagrams for Distributed Systems. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pages 310–318, 2016.