Mouakher I, Dhaou F, Attiogbé JC. Event-based semantics of UML 2.X concurrent sequence diagrams for formal verification. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(1): 4–28 Jan. 2022. DOI 10.1007/s11390-021-1673-5

Event-Based Semantics of UML 2.X Concurrent Sequence Diagrams for Formal Verification

Inès Mouakher[1], Fatma Dhaou[1], and J. Christian Attiogbé[2]

[1] *Faculty of Sciences of Tunis, University of Tunis El Manar, Tunis 1068, Tunisia*
[2] *Institute of Technology, University of Nantes, Nantes 44 322, France*

E-mail: {ines.mouakher, fatma.dhaou}@fsegt.utm.tn; christian.attiogbe@univ-nantes.fr

Received June 1, 2021; accepted November 16, 2021.

Abstract UML 2.X sequence diagrams (SD) are among privileged scenarios-based approaches dealing with the complexity of modeling the behaviors of some current systems. However, there are several issues related to the standard semantics of UML 2.X SD proposed by the Object Management Group (OMG). They mainly concern ambiguities of the interpretation of SDs, and the computation of causal relations between events which is not specifically laid out. Moreover, SD is a semi-formal language, and it does not support the verification of the modeled system. This justifies the considerable number of research studies intending to define formal semantics of UML SDs. We proposed in our previous work semantics covering the most popular combined fragments (CF) of control-flow ALT, OPT, LOOP and SEQ, allowing to model alternative, optional, iterative and sequential behaviors respectively. The proposed semantics is based on partial order theory relations that permit the computation of the precedence relations between the events of an SD with nested CFs. We also addressed the issue of the evaluation of the interaction constraint (guard) for guarded CFs, and the related synchronization issue. In this paper, we first extend our semantics, proposed in our previous work; indeed, we propose new rules for the computation of causal relations for SD with PAR and STRICT CFs (dedicated to modeling concurrent and strict behaviors respectively) as well as their nesting. Then, we propose a transformational semantics in Event-B. Our modeling approach emphasizes computation of causal relations, guard handling and transformational semantics into Event-B. The transformation of UML 2.X SD into the formal method Event-B allows us to perform several kinds of verification including simulation, trace acceptance, verification of properties, and verification of refinement relation between SDs.

Keywords UML 2.X sequence diagram, formal semantics, nested combined fragment, partial order theory, Event-B

1 Introduction

With the increasing complexity of current distributed systems, their modeling, analysis and verification become more and more complex. Formal methods, based on mathematical foundations, contribute to improving the criteria of reliability and integrity. However, they remain difficult to adopt in early stages of modeling, where scenario-based approaches are widely used since they are more intuitive.

UML 2.X sequence diagrams (SDs) have gained increased popularity and have become a de facto standard scenario-based language for modeling interactions of systems. The interactions describe the behavior of a system with the exchanged messages between its components, which are modeled by lifelines. The semantics of interactions is usually described with traces composed of event occurrences. Interaction specifications are partial ordering to constrain allowed and disallowed traces; combined fragments (CFs) can be used to allow a compact syntactic representation of the interactions or alter the orderings of occurrences of the events. In the standard specification of SDs and in the most reviewing approaches, proposed semantics refer only to the valid traces because they describe the behavior of modeled systems; however, invalid traces describe intuitively some system properties or counter examples.

Although the Object Management Group (OMG)[①]

Regular Paper
Special Section on Software Systems 2021—Theme: Dependable Software Engineering

① OMG Unified Modeling Language, Version 2.5.1, 2017. https://www.omg.org/spec/UML/2.5.1/PDF, Oct. 2021.

has defined an official standard semantics for UML 2.X SDs, some shortcomings still persist, which explains the diversity of the studies [1–6] proposing various kinds of semantics to remedy with them. However, none of them is complete even if we consider only valid interactions defined by basic SDs or by the most popular CFs: ALT, OPT, LOOP, SEQ, STRICT and PAR modeling alternative, optional, iterative, sequential, strict and parallel behaviors respectively. These issues are mainly due to the fact that the standard semantics is informal; thus the interpretation and analysis of the SDs is ambiguous. Indeed, numerous challenges connected with making semantics for sequence diagrams and choices must be made, such as: 1) computing the partial order relation between the occurrence specifications (i.e., the events); 2) the asynchronous communication between the events; 3) the presence of multiple CFs, their composition (with WEAK SEQUENCING) and their nesting; 4) the ambiguities related to the evaluation of the interactions constraints (i.e., guards) for some combined fragments; for instance the ambiguities for ALT CF are the exact time of the evaluation, which lifeline must evaluate the guard, which operand to choose when several guards are true, etc.; and 5) the synchronization between components of the system after the guard evaluation.

Our work aims at improving the coverage of the formal verification of UML-based models. In our previous work, we proposed an interpretation semantics of UML 2.X SDs [7–9] that gives a close interpretation of the standard description of the UML 2.X sequence diagrams while admitting more valid traces compared with the standard interpretation. Indeed we have proposed: 1) an intuitive formalization of UML 2.X SDs, covering the most popular CFs (SEQ, ALT, OPT and LOOP) as well as their composition and their nesting, and rules for the computation of the partial order relation between events [7]; 2) a solution for guard evaluation and synchronization issues [8]; 3) an operational semantics [9] which is independent of any target formalism. We now propose additional contributions. We first extend our interpretation semantics by: 1) generalizing the formalization of UML 2.X SDs for covering more important CFs (PAR and STRICT) as well as their composition and their nesting, and 2) defining new rules for the generalization of the previous ones for both the computation of precedence relations between the events of UML 2.X SD, and the solution for guard evaluation and synchronization issues.

UML 2.X SD is a semi-formal language, and the analysis, the expression and the verification of important properties of the modeled system can be done by the coupling with formal methods offering powerful tools dedicated to these goals. Event-B [10,11] is a well-known formal method, offering a wide range of verification tools (theorem provers and model checkers). The second contribution of this paper consists in proposing a transformational semantics for UML 2.X SD with nested guarded CFs, modeling the behaviors of distributed systems, in Event-B [10] formalism. Indeed, Event-B, as a modeling method for discrete transition systems, allows to capture intuitively the proposed interpretation semantics of SDs, since it uses the set theory as modeling notation. Also, it offers an available toolset that enables us to verify trace acceptance, well-formedness properties of SDs, to validate properties expressed in LTL[e]② (standing for linear temporal logic), as well as the animation and simulation of the behavior of the SD. Finally, Event-B embeds a fundamental concept: refinement allowing the stepwise development of complex systems, which deserves for our ongoing work that is the checking of the correctness of refinement relation between SDs.

The paper is organized as follows. Section 2 introduces the main concepts related to this work. Section 3 discusses related work and presents the key features of our approach. Section 4 introduces the interaction language, its behavior described by event structure and the computation of partial order relation. Section 5 describes our transformational semantics in Event-B. Section 6 presents the verification and validation of UML 2.X SDs and with Rodin/ProB frameworks [11,12]. Finally, Section 7 provides concluding remarks and perspectives.

2 Preliminaries

2.1 Brief Introduction to Event-B

Event-B [10] is a formal method used in numerous industry projects and in academia [13]. It is dedicated to the modeling of critical systems. The basis for the mathematical language in Event-B is the first-order logic and the typed set theory. The set-theoretical notation of Event-B defines different kinds of relations and functions enhanced by different properties. The Event-B method is supported by the Rodin toolkit [11] which comprises editors, theorem provers, animators

② https://prob.hhu.de/w/index.php?title=LTL_Model_Checking, Jan. 2022.

and model checkers (ProB) [12]. In addition, Event-B provides flexible refinement techniques.

Table 1 illustrates some Event-B notations, where S and T are sets, and R is a relation. Intuitively, a binary relation from S to T is a set of mapping (x, y) where $x \in S$ and $y \in T$, a many-to-many mapping. A partial function is a relation where each element of the domain is related at most to a unique element in the range, which is a many-to-one mapping. A total function is a partial function from S to T where each element x in S has exactly one mapping in the range. A surjection is a function from S to T which maps all elements in T. A bijection is a surjection and one-to-one mapping.

Table 1. Examples of Event-B Notations

Name	Notation
Cartesian product of sets S and T	$S \times T$
Power set of set S	$\mathbb{P}(S)$
Binary relation symbol	$\leftrightarrow$
Bijective function symbol	$\rightarrowtail$
Total surjection symbol	$\twoheadrightarrow$
Irreflexive transitive closure of R	R^+
Partial function symbol	$\nrightarrow$
Range of relation R	$ran(R)$

2.2 UML 2.X Sequence Diagrams

In the OMG specification, SD is introduced as the most common kind of interaction. The syntax of SD is presented by graphical notations and by a metamodel.

Fig.1 illustrates an example of an interaction. The lifelines represent the individual participants in the interaction, which interact with messages. For each message m, there are two events $!m$ and $?m$ that denote the send and the receive events of the message m respectively. CFs allow the modeling of sophisticated interactions. We consider the following CFs, SEQ, ALT, OPT, PAR, STRICT, and LOOP. Table 2 summarizes the description of each of them according to the standard semantics. The constituent CFs can be nested, and the WEAK SEQUENCING③ operator is the default composition operator for CFs.

2.3 Motivating Example

Fig.2 illustrates a concurrent access by an administrator and a client to an application server. The system is composed of the components Admin, Client, Application Server, Master Database Server, Database Server1 and Database Server2. The administrator can connect to perform some specific administrative tasks. In this SD, the administrator requests to the application server the list of the queries, and then the server transmits this request to the master database server. This latter returns the list of the queries to the application server, which displays the list to the administrator. At the same time, a client can connect to the application server through a specific profile, and then it inputs a query. The application server processes the query; two alternatives are possible: if the query is success fully found, the server transmits the result to the client; otherwise it displays the message "not_found". The server can do at most two searches for a query. The server can iterate the processing of the query at most three times.

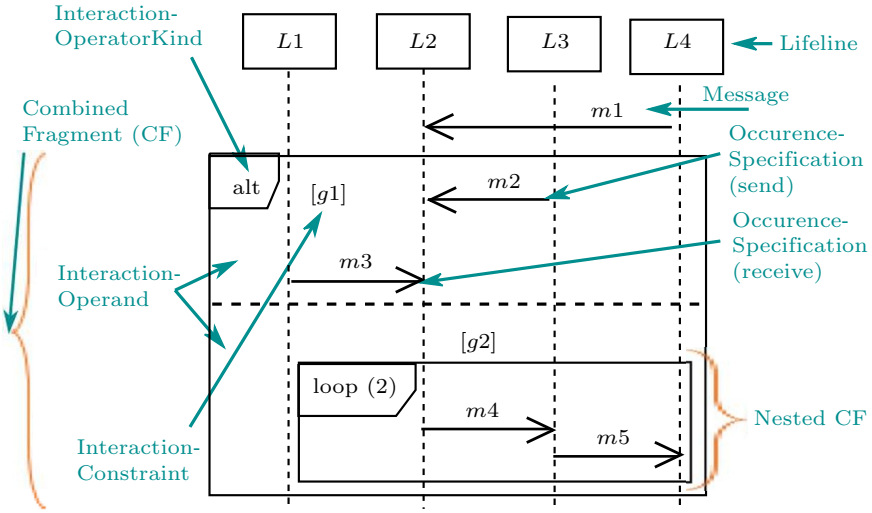

Fig.1. Example of sequence diagram.

③See Table 2 for the definition of WEAK SEQUENCING.

Table 2. Description of CFs According to OMG Standard[4]

CF	Description
SEQ	SEQ CF represents a WEAK SEQUENCING between the behaviors of the operands. Weak sequencing is defined by the set of traces with these properties: 1) the ordering of OccurrenceSpecifications (occurrences of the events) within each of the operands is maintained in the result, 2) OccurrenceSpecifications on different lifelines from different operands may come in any order, and 3) OccurrenceSpecifications on the same lifeline from different operands are ordered such that an OccurrenceSpecification of the first operand comes before that of the second operand.
ALT	ALT CF represents a choice of behavior. At most one of the operands will be chosen.
OPT	OPT CF permits to model a choice of behaviors where either the (unique) operand happens or nothing happens. An OPT CF is semantically equivalent to an ALT CF with two operands such that the first one has a non-empty content and the second one is empty.
PAR	PAR CF represents a parallel merge between the behaviors of the operands. The occurrence of the events of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.
LOOP	LOOP CF permits to model an iterative behavior. The LOOP operand will be repeated a number of times. The guard may include a lower (min) and an upper number (max) of iterations of the LOOP as well as a boolean expression. The semantics is such that a LOOP will iterate at least the minimum number of times (given by the iteration expression in the guard) and at most the maximum number of times. After the minimum number of iterations are executed and the boolean expression is False, the LOOP will terminate.
STRICT	STRICT CF designates a strict sequencing between the behaviors of the operands. The semantics of strict sequencing defines a strict ordering of the operands.

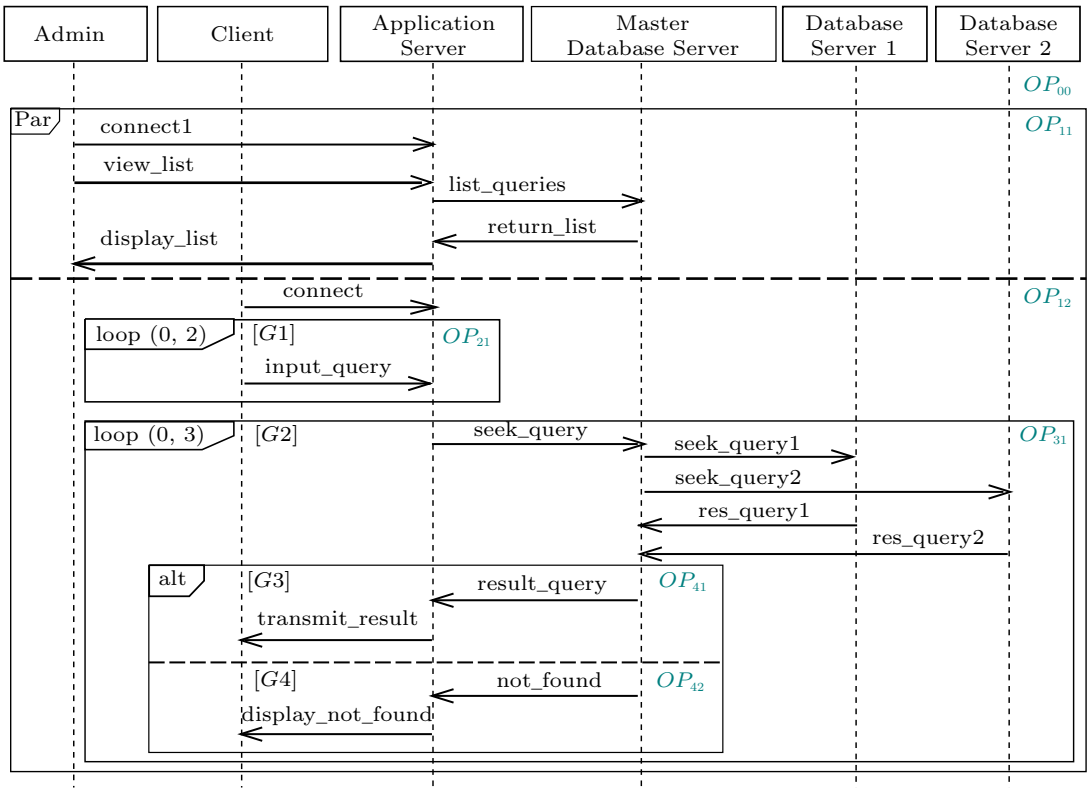

Fig.2. SD of interaction with concurrent access to a database.

3 Discussion and Related Work

Subsection 3.1 presents and discusses the informal semantics defined in the OMG specification. In Subsection 3.2, we discuss some approaches supporting algorithms and tools for the verification of SDs. In Subsection 3.3, we present the key features of our approach.

3.1 Interpretation in Standard Semantics

The UML 2.X SDs standard definitions and semantics are informal and ambiguous. This leads to several

[4]OMG Unified Modeling Language, Version 2.5.1, 2017. https://www.omg.org/spec/UML/2.5.1/PDF, Oct. 2021.

subtle choices in the interpretation of the language constructs. These variations points are intentional to adapt its use according to the application field. The standard proposes trace-based semantics for SDs, where the trace is defined as a set of occurrence specifications. The trace semantics is a very intuitive semantics; however it leads to exponential blow-up of the representation that is accentuated in the context of distributed systems.

Interpretation of Basic Interactions. The standard semantics of UML 2.X SDs interprets the order of events on each lifeline by reading the diagram from top to bottom. In context of distributed systems, the objects are independent, the communication is asynchronous and in the general model of communication, the reception of messages is not ordered. Then, imposing a total order of the events in each lifeline leads to the loss of some possible valid behaviors that causes the emergence of unspecified behaviors in the implementation. In Fig.1, we consider messages $m1$, $m2$ and $m3$ according to the rules of the standard semantics: the events $!m1$, $!m2$ and $!m3$ on $L4$, $L3$ and $L1$ lifelines respectively are not ordered; even if $L4$, $L3$ and $L1$ are independent, on the $L2$ lifeline the events $?m1$, $?m2$ and $?m3$ are ordered. The constraint of the total ordering of the events on each lifeline can be relaxed by the use of the COREGION operator, which is applied on one lifeline, and permits an implicit parallelism⑤. However, in some cases (see Fig.3), the COREGION operator cannot solve the issue of the total ordering of the events along a lifeline, and additional messages must be added to restore some precedence relations between some events [2].

As a result we have an overcrowding of the graphical representation due to the overlapping between the existing combined fragments (CFs) and the added COREGION operator. This leads to difficulties for the delimitation of the beginning and the end of each COREGION operator, especially if we have several nested CFs.

Combined Fragments. We focus especially on a subcategory of CFs: ALT, OPT, LOOP, SEQ, STRICT and PAR, allowing to model alternative, optional, iterative, sequential, strict and parallel behaviors respectively; the first four CFs permit a compact syntactic representation of behaviors, and the last ones alter the orderings of occurrences of the events [1]. The standard does not specify synchronization points at the entering and exiting of a CF. To illustrate this, let us consider the example in Fig.1. Message $m1$ is located above the ALT fragment, but there is no precedence relation between the event $!m1$ and the events $!m2$ and $!m3$: $!m1$ can occur independently of events $!m2$ and $!m3$. The standard semantics is "compositional in the sense that the semantics of an interaction is mechanically built from the semantics of its constituent interaction fragments. The constituent interaction fragments are ordered and combined by the WEAK SEQUENCING operator." The LOOP operator is defined as a recursive application of the SEQ operator, and then there is WEAK SEQUENCING between the iterations of a LOOP (see Fig.4). In ALT CF, at most one of the operands will be non-deterministically chosen from its operands that have implicit (or explicit) true guards; then an ALT operator offers more expressive power than an IF operator in programming languages. Multiple CFs in an SD and their nesting further complicate the determination of precedence relations between events. In the standard semantics, an arbitrary nesting of CFs is allowed. Nevertheless, the combination of some kinds of CFs is very complex, and it may lead to hidden issues for some combinations.

Interaction Constraints (*Guards*). It may be contained in some operands and influence the computation

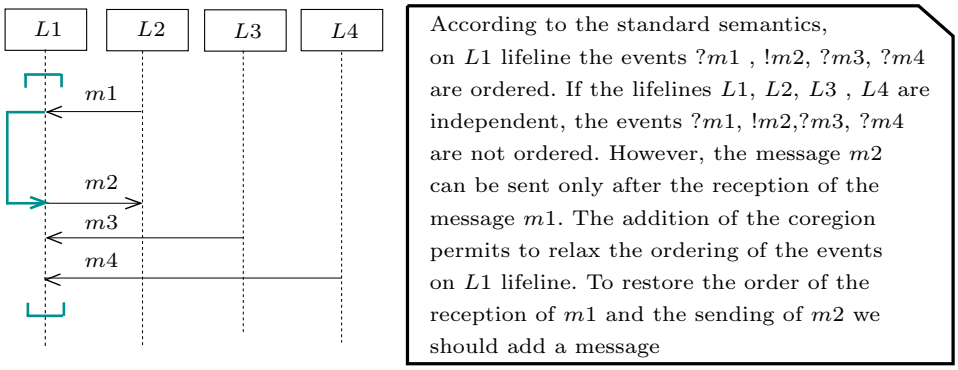

Fig.3. Basic SD with the addition of COREGION.

⑤The covered events by the COREGION operator occur in any order.

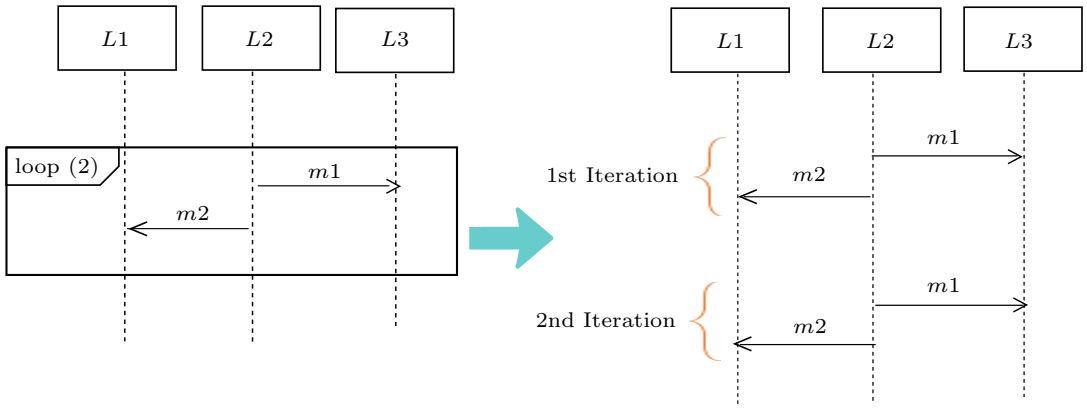

Fig.4. Processing of an SD with LOOP CF. With the weak sequencing, we can have: the sending events of the messages $m1$ and $m2$ of all the iterations can occur, and then the receiving events come after. With the strict sequencing, all the sending events and all the receiving events of the first iteration must occur, and then the events of the second iteration can occur.

of partial order relations between events. For instance in the ALT CF composed of several guarded operands, all guards can be evaluated to false; hence the CF is omitted. Moreover we can have one, or several guards evaluated to true. The unique mention in the standard semantics of guard is the following: "guard has to be evaluated before the occurrence of the first event of each operand". This is problematic, since the first event of an operand has not been clearly defined. Moreover, guards can be placed on different lifelines, and several guards can be true simultaneously. In Fig.1, the first operand of ALT CF has two possible first events: $!m2$ on the $L3$ lifeline and $!m3$ on the $L1$ lifeline. The second operand has one first event: $!m4$ on the $L2$ lifeline. This illustrates a non-local choice among the $L1$, $L2$ and $L3$ lifelines. Another aspect that should be considered after the guard evaluation, is the synchronization issue between the lifelines. We should deal with the synchronization correctly to preserve the standard meaning of the ALT CF and the LOOP CF. For the ALT CF and the LOOP CF, the synchronization must be ensured between the lifelines respectively, once the choice of an operand among several potential operands is made, and once the guard changes its value during the iterations and becomes false.

Processing the Diagram. To master the complexity of partial order's computation between the events of the SDs with CFs, the UML semantics adopts a preprocessing that consists in transforming the SD with CF into basics SD by flattening the CF. The partial order between events of each construct is computed independently, and then they are composed by the WEAK SEQUENCING operator. Note that usually, the intermediate representation of SDs is not given in detail, which introduces many possible variations of interpretation. Furthermore, with this, pre-processing the benefits of the compact syntactic graphical representation is lost.

3.2 Reviews of Existing Semantics

In the literature, there are a considerable number of research studies intending to verify UML diagrams. Existing methods dealing with the sequence diagrams focused on various purposes: the checking of properties, consistency of multi-view models (MVM), containment and refinement relationship among sequence diagrams, trace acceptance, etc. Table 3 summarizes some existing studies offering tools allowing the verification of SDs. We distinguish tools based on trace analysis and model checking tools (such as SPIN, nuSMV, CADP, and so on). All the cited approaches in Table 3 consider basic interactions based on the standard interpretation (SEQ). The third column (sem.) indicates the proposed semantics by authors, and we use the acronyms MT, Op. and Den. to indicate Model Transformation, Operational and Denotational respectively. The "SD Concepts" column indicates the SD concepts that we cover: ALT, LOOP, STRICT, PAR, guard and nesting. ✓indicates that the feature is covered by the work and × indicates it is not. Otherwise, the feature is covered with some restrictions: Det. indicates determinist choice for an ALT CF, FL for finite loop (i.e., without guard), SL for strict sequencing between loop iterations, Const. indicates the guard is defined as constraint (nonstandard guard definition), Global for global evaluation of guards, SP for synchronization point between CFs, and WR for with restriction, i.e., not all nesting of CFs is allowed. Most of the existing formalizations are incomplete such that they consider

Table 3. Comparison of Approaches Supporting the Verification of Sequence Diagram

	Reference	Sem.	SD Concepts						Tool	Purpose	Formalism
			ALT	LOOP	STRICT	PAR	Guard	Nesting			
Trace analysis	Lund (2007) [4]	Op.	✓	FL	✓	✓	Const.	✓	Escalator	Refinement Test	Maude
	André *et al.* (2017) [5]	MT	Det.	×	×	✓	✓	SP	✓	Test	TERMOS
	Mahe *et al.* (2020) [6]	Op. Den.	✓	✓	✓	✓	×	✓	Python script	Tr. acceptance	Set of rules
Model checking	Rasch and Wehrheim (2005) [14]	MT	×	×	×	×	✓	×	FDR	Consistency of MVM	CSP
	Knapp and Wuttle (2007) [15]	MT	✓	✓	✓	✓	Global	WR	HUGO/RT	V. Properties	Automata
	Lima *et al.* (2009) [16]	MT	Det.	FL	×	✓	Global	✓	SPIN	V. Properties Simulation	PROMELA LTL
	Cunha *et al.* (2011) [17]	MT	×	×	×	×	×	×	MCKit & nuSMV	V. Properties	PN CTL
	Zhu *et al.* (2012) [18]	MT	✓	SL	×	✓	✓	✓	Algorithms	V. Properties	TTS4SDT
	Miyazaki *et al.* (2012) [19]	MT	✓	✓	✓	✓	×	×	LTSA	V. Properties Refinement	FSP LTL
	Remenska *et al.* (2014) [20]	MT	×	×	×	×	×	×	mCRL2	V. Properties	μ-calculus LTL
	Muram *et al.* (2019) [21]	MT	✓	✓	✓	✓	Global	✓	NuSMV	Containment	SMV LTL
	Lima *et al.* (2016) [22]	MT	Det.	SL	✓	✓	Global	✓	FDR3	Consistency of MVM	CML
	Pan *et al.* (2019) [23]	MT	Det.	✓	×	×	Global	SP	ISDChecker SpaceEx	V. Properties	IA
	Chen *et al.* (2020) [24]	MT	✓	✓	✓	✓	Global	SP	MyCCSL	V. Properties Simulation	CCSL
	Ours	MT	✓	✓	✓	✓	✓	✓	ProB	V. Properties Tr. acceptance Animation Simulation Refinement	Event-B LTL

only the subset of CFs or make assumptions that depart from the standard interpretation of sequence diagrams. In the following, we categorize different choices taken mainly by the approaches introduced in Table 3. In particular, we discuss how these approaches dealt with basic SDs and on the more popular CFs: ALT, OPT, LOOP, SEQ, STRICT and PAR.

Semantics of SD. Several previous studies have proposed approaches for applying model transformation (MT) from UML SDs to formal formalisms such as Integration Automata (IA) [23], Petri Nets (PN) [17], COMPASS Modelling Language (CML) [22], Finite State Processes (FSP) [19], Clock Constraint Specification Language (CCSL) [24], Symbolic Model Language (SMV) [21], Timed Transition System for SDT (TTS4SDT) [18], Communicating Sequential Processes (CSP) and Linear Temporal Logic (LTL) [21]. Although these translations allow reusing advantageously the target formal formalism's tools, relying on them to capture semantics of SDs sometime leads to imposing restric-

tions of the SD interpretations and reasoning on foreign concepts. There are a few approaches proposing tools based on Operational (Op.) and Denotational (Den.) semantics for SDs. These approaches offer a rather interesting interpretation of SDs; however they propose transformation rules from the proposed semantics into the language of the target tools and they usually support a subset of the proposed semantics. Compared with transformational semantics, these approaches are based on more complicated process. In [4], the authors defined, first, an operational semantics based on an execution system. Then, the operational semantics is implemented by the use of rewriting rules in Maude that implement the rules of both the execution system and the projection system. In the Maude implementation, the guards are not supported and some restrictions are made for the LOOP operator. In [6], the authors proposed denotational semantics that serves as mathematical foundation and operational semantics. A trace analysis tool is adapted from the Op. semantics.

Partial Order Relation Between Events. Most of the existing approaches are based on the definitions of the standard semantics for the computation of traces that imposes the total ordering of the events on each lifeline. Only few approaches propose a different partial order relation between events to interpret SDs. For example in [25], the authors showed that the partial order relation between events depends on the considered architecture. Different partial orders defined are visual order, enforced order (i.e., the order of the events that the underlying architecture can guarantee to occur), and inferred order. In [26], the authors proposed causal semantics for basic SDs modeling behaviors of distributed systems. They defined rules for the computation of partial order relations taking into account the independence between the components (modeled by lifelines) involved in the interactions. Even if in UML we can alter the orderings of occurrences of the events by the use of the COREGION or PAR constructs, adopting a reading semantics customized with respect to the type of the studied systems allows to alleviate the representation of the diagram, and it remains easier and more intuitive for the specifier.

Combined Fragments. Introducing CFs disturbs the interpretation of the SDs and complicates the computation of the partial order between the events. To ease the processing of the SDs with CFs, some of the existing approaches do not properly deal with some CFs. Another challenge with CFs is the consideration of nesting CFs. Indeed, the combination of some kinds of CFs is very complex, and it may lead to hidden issues. To avoid ambiguous cases resulting from nesting CFs, the existing studies [5, 15, 23, 24] adopt some assumptions, for instance by limiting the depth of nesting CFs, and by considering the combination of a small number of CFs. Some of them do not consider CFs and deal only with basic SDs [14, 17].

Due to the weak sequencing, events that do not belong to the same CF can occur independently and they overlap. This generates non-intuitive behaviors. To avoid this issue many approaches propose to restrict weak sequencing. On the one hand, this restriction increases causal dependency between the events, which leads to the loss of some possible traces; on the other hand it considerably limits the expressive power of the language. Other approaches do not support the composition or nesting of CFs. In [24], the authors defined transformation rules for formally representing constructs of SDs in the CCSL constraints. They did not specify how the constructs of SDs can be composed or nested. They specify synchronization points for ALT, and LOOP CFs. In [20], the authors presented a property assistant wizard (PASS) as part of a UML-based front-end to the mCRL2 toolset: the tool supports μ-calculus, which is an event-based formalism, making it a good match for SDs; however the authors had not formally given the rules of this matching. The approach proposed in [19] integrates multiple SDs using hMSC (high-level Message Sequence Chart) into a SD. Then, the diagram is translated into Finite State Processes (FSP) representation to support model checking with the Labeled Transition System Analyzer (LTSA) tool. The authors did not give the interpretation to the composition or nesting of CFs. The translation of a CF consists in defining actions at its beginning and end.

Some approaches adopt a non-standard and restrictive interpretation that forces the lifelines to synchronize at decision points of ALT, OPT and LOOP CFs. In [23], the authors considered only three CFs, which are LOOP, ALT and OPT. They enforced strict sequencing on the fragments and they imposed that a fragment covers all lifelines. Hence, when the execution control of flow enters a fragment, all lifelines enter the fragment. In [5], the authors interpreted the entering and exiting of a CF as a synchronization point for the participating lifelines. In [22], the authors proposed translation rules that are recursively applied (top-down) into the COMPASS Modelling Language (CML). They translated CFs by imposing the synchronization point on each CF. In [21], the authors used the model checker

NuSMV to verify the containment relationship between SDs. They translated both high-level and low-level SDs into temporal logic based constraints (LTL) and symbolic model language (SMV) respectively. They defined the ALT operator as an if-then-else statement, and they imposed a strict sequencing between the iterations of LOOP CFs, and restricted the use of weak sequencing loops to contain only a basic interaction. In [15], the authors restricted the LOOP CF to contain only a basic interaction and they proposed a new construct (SLOOP) that considers strict sequencing of LOOP iterations. In [18, 22], strict sequencing is used between the successive iterations of LOOP.

Interaction Constraints. In the literature, the guard evaluation issue is handled in different ways [1]. Similarly, to the guard evaluation, the synchronization issue is addressed in several ways by existing studies that adopt sometimes some restrictions which limit the expressive power of these CFs. For instance in some approaches [3, 6, 19], guards are not explicitly evaluated. In [15, 16, 21–23], the guard is evaluated globally. In [22], a guard is evaluated by all lifelines. This approach can raise an issue if a guard is evaluated at different times. In TERMOS [5], the authors imposed a global time point when all participating lifelines evaluate the guards and choose one alternative (this will be represented by a common transition in the formal semantics). Moreover, only a deterministic form of guarded choice is allowed (similar to an if-then-else construct). In [16], the guard variable is globally declared to enforce all lifelines to get the same decision at the choice point. The LOOP CF works only with a fixed number of repetitions. In [4], resolving operators (e.g., choosing the branch of an ALT) are considered by silent events. Furthermore, rules and special events are defined to handle guarded choices. Guarded choices are defined as special applications of the local constraint construct (i.e., local to a lifeline); thus a trace with a false guard is invalid. The LOOP is considered by flattening it recursively with the SEQ operator and a counter is used and decreased by 1 at each iteration.

3.3 Key Features of Our Approach

Our approach brings together two semantics for UML 2.X SDs. Indeed we propose an appropriate semantics of interpretation for UML 2.X SDs with nested guarded CFs allowing the computation of partial order relations. Based on this semantics, we propose a transformational semantics, into Event-B [10].

The main advantage of this approach compared with existing approaches in the literature to formal semantics for UML, is two-fold. First, the computation of partial order relations between events induces a weaker scheduling constraints, and supports SDs with guarded nested CF; it results in more expressive SDs, as each SD describes a larger number of acceptable behaviors. Second, the transformational semantics allows us to enable several kinds of analysis and verification on SDs including simulation, trace acceptance, verification of properties, verification of refinement relation between SDs, and so on. Indeed, with the Event-B tools Rodin and ProB [12], we support different types of the SD verification while existing approaches allow the verification of one or two types.

Computation of Causal Relations. We extend and generalize our previous work [7], where we introduced a new formal definition of UML 2.X SD based on set theory. As interpretation semantics, we have considered a kind of existing semantics named causal semantics. It was proposed by [26] for basic SDs modeling behaviors of distributed systems. Its rules, permitting the computation of partial order relation, take into account the independence of the components involved in the interactions. This high expressive power facilitates the task of the designer since a great number of cases can be described, and it prevents the issue of the emergence of unspecified behaviors in the implementation. The rules of causal semantics for basic SDs, as defined in [26] are: 1) synchronization relationship $<_{Sync}$ ensures that each message m is received only if it was previously sent; 2) reception-emission relationship $<_{RE}$ ensures that receiving a message causes the sending of the message that is directly consecutive to it; 3) emission-emission relationship $<_{EE}$ guarantees that if two messages are sent by the same lifeline, their sending events are ordered; 4) reception-reception relationship $<_{RR}$ ensures that in the case where two messages sent by a lifeline l are addressed to the same lifeline l': if they are emitted in this order by l, they are treated in the same order by l'. Then, the causal order relation $<_{caus}$ is defined as follows: $<_{caus} = <_{Sync} \cup <_{RE} \cup <_{EE} \cup <_{RR}$. The event occurrence depends on the partial order relationship $<_{caus}$. In Fig.2, we consider the reception of the messages res_query1 and the reception of the message res_query2 on the Master Database Server lifeline, according to the definitions of the standard semantics these events are ordered; however in the considered semantics they are not.

Guard Handling. We extend our previous work [9]

to deal with guards. Fictitious events (τ events) are used to evaluate guards as well as to ensure the synchronization between the lifelines that are covered by the considered operands. Our approach is appropriate in the context of SD modeling behaviors of the distributed system, since the components of distributed systems are independent and some special events are needed to synchronize them. We propose an approach for the guard evaluation that is faithful with the standard semantics. For each guarded operand of a CF (for instance ALT or LOOP) we associate a positive fictitious event when the guard is evaluated to True, and a negative fictitious event when guard is evaluated to False. Each fictitious event belongs to the same lifeline of the first event of the considered operand; it is placed above the first event. The guard of an operand is evaluated one time by the fictitious event. For the SD in Fig.2, we introduce two fictitious events for each guarded operand of the set $\{OP_{21}, OP_{31}, OP_{41}, OP_{42}\}$.

Transformational Semantics in Event-B. The main purpose of the definition of formal semantics is to implement it and automate the analysis and the verification process. The transformational semantics is more simple to implement since the semantics is defined with the input language of used tools. Our transformational semantics is based on Event-B. The Event-B method is dedicated to describing event-driven reactive systems; it allows us to describe intuitively interactions. We take advantage of the Event-B method refinement, which permits to support refinement between SDs. The Event-B method and its tools support (Rodin/ProB) are used to specify, animate, verify and simulate our semantics.

4 Formalization of Sequence Diagram

The event-based semantics of an SD consists in expressing and ordering all the events of this SD.

The events are associated with messages exchanged between the components interacting in the SD. For each message of the SD, we associate a sending and a receiving event. Moreover, we consider fictitious events that are required for guard handling and synchronization between lifelines. Our semantics is based on the causal relationships allowing the computation of partial order relations between events of the SD. In Subsection 4.1, we introduce the formal definition for UML 2.X SDs. In Subsection 4.2, we introduce the event-based semantics of SD as labeled transition systems (LTS). In Subsection 4.3, we define the rules for computing the causal relationships.

4.1 Interaction Language

In this subsection, we propose an abstraction of SD with a nested CF in the form of tree, and then we give formal definitions based on set theory and tree structure. Moreover, we have defined different functions and relations that allow the navigation in this structure and the determination, straightforwardly, of the required relationships between the operands. We consider a subset of SDs containing CFs of control-flow ALT, OPT, LOOP, SEQ, PAR and STRICT CF. They can be nested or sequential. We assume that the operands of the CF do not overlap, but can be nested. Our approach supports multiple nesting levels of fragments. The UML standard defines the graphical notation of SDs and the abstract syntax of the UML standard. We introduce formal definition for SDs. It is suited as basis for defining semantics, and it is abstract and contains only information required for the semantics and the hypotheses on the considered SDs. We use the same set theory notation as that of the Event-B method. We introduce the following definition for SDs.

Definition 1 (Sequence Diagram). *A sequence diagram $\mathcal{SD}$ is a tuple $\langle L, M, <_{\mathcal{SD},L}, OP, F, tree\_OP\rangle$, where:*

- *L is a non-empty set of lifelines,*
- *M is a non-empty set of asynchronous messages,*
- *$<_{\mathcal{SD},L} = \bigcup_{l_i \in L} <_{\mathcal{SD},l_i}$ is the union of the partial order relation between messages on each lifeline l_i,*
- *OP is a set of operands,*
- *F is the set of combined fragments, and*
- *$tree\_OP$ is a partial function that allows to structure the $\mathcal{SD}$ in the form of a tree of operands.*

4.1.1 Operands and Combined Fragments

The general definitions of operands and combined fragments are given as follows.

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

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

where k is the number of operands in the CF F_i, $guard_{ij}$ is the guard of the operand OP_{ij}, $weight_{ij}$ is the weight of the operand OP_{ij}, and M_D_{ij} is the messages that are directly contained in the operand OP_{ij}.

Each operand in an SD has a weight. For instance, each operand of SEQ, ALT, OPT, PAR or STRICT CF has a weight equal to 1; an operand of a LOOP CF has a

weight equal to a value of the maximum number of iterations of the considered LOOP CF. For an operand of ALT CF, we reinforce the guard $guard_{ij}$, by adding the constraint that none of the other operands is chosen.

Definition 3 (Combined Fragments). *We define a set of combined fragments, $F = \{F_1, F_2, ..., F_n\}$ is the set of n combined fragments, $F_i = \langle OP_i, operator_i, L_i \rangle$, where OP_i is a set of operands, $operator_i$ is an operator, and L_i is the set of lifelines that are covered by the combined fragment. The operator $operator_i$ is one of the operators* SEQ, ALT, OPT, PAR, STRICT, *and* LOOP.

4.1.2 Abstract Structure for Sequence Diagrams

The tree structure is well-suited to capture the nested structure of SDs. A non-empty tree is a root node connected to 0 or more sub-trees. The descendants of a node n are all nodes reached from n to the leaf nodes. A path is a sequence of nodes, where there is a unique edge from one node to another. The path can be only downward and it connects a node with a descendant. The length of a path is the number of edges in the path. The depth of a node n is the length of the path from the root to node n. The ancestors of a node n are all nodes found on the path from the root to node n.

We associate a label to each operand. Two operands with the same first part of the index (prefix i) belong to the same combined fragment; for instance, in Fig.2, OP_{41} and OP_{42} belong to the same ALT CF. 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 CFs. Fig.5 illustrates the associated tree for the SD in Fig.2.

We define the tree structure for SD operands as follows [10, 27].

Definition 4 (Tree Structure for SD Operands). *The tree structure related to an SD is defined by the root operand OP_{00} and the total function $tree\_OP$ mapped from all operands except OP_{00} to their parent operand. The function $tree\_OP$ satisfies these properties: acyclic, non-reflexive and connectivity (i.e., all operands are connected to the root). Formally, the tree structure is defined as*:

$$\begin{cases} OP_{00} \in OP, \\ tree\_OP \in OP \setminus \{OP_{00}\} \to OP, \\ (tree\_OP)^+ \cap id(OP) = \emptyset, \\ (OP \setminus \{OP_{00}\}) \subseteq ((tree\_OP)^+)^{-1}[\{OP_{00}\}]. \end{cases}$$

The third line of the formal definition expresses the acyclic and non-reflexive properties, since the intersection between the transitive closure of the $tree\_OP$ function and the identity function is an empty set, and thus there are no loops in the tree structure. The last line guarantees the connectivity property, and since $((tree\_OP)^+)^{-1}[\{OP_{00}\}]$ gives all operands connected to the root operand OP_{00}, then all operands except the root are connected to the root. We consider the following functions.

- *anc* gives a set of ancestor operands.

$$\begin{cases} anc : OP \to \mathbb{P}(OP), \\ anc(X) = tree\_OP^+(X). \end{cases}$$

- LCA is a function that gives the first ancestor in common for two operands:

$$\begin{cases} LCA : (OP \times OP) \to OP, \\ LCA(X, Y) = X, \text{if } X = Y \text{ or } X \in anc(Y), \\ LCA(X, Y) = Y, \text{ if } Y \in anc(X), \\ LCA(X, Y) = LCA(tree\_OP(X), tree\_OP(Y)), \\ \qquad \text{otherwise.} \end{cases}$$

- *weight* is a function that gives the weight of each operand: $weight : OP \to NAT^+$. For example in Fig.2

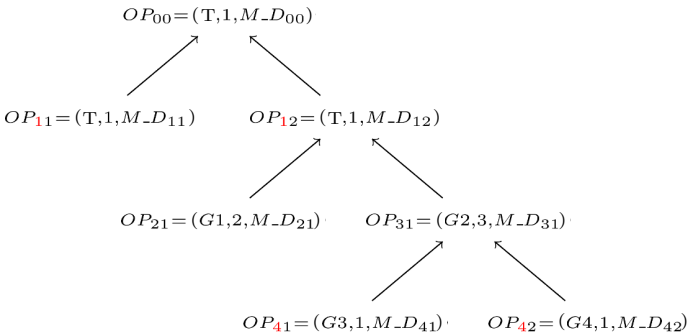

Fig.5. Tree associated with the SD in Fig.2. T: true.

$weight(OP_{12}) = 1$ and $weight(OP_{31}) = 3$. We overload the function $weight$ to associate the weight of the path between two operands X and Y, where Y is a descendant of X.

$$\begin{cases} weight : (OP \times OP) \to NAT^{+}, \\ weight(X,Y) = 1, \text{ if } X = Y, \\ weight(X,Y) = weight(Y) \times \\ \quad weight(X, tree\_OP(Y)), \\ \quad \text{if } Y \in anc(X). \end{cases}$$

For example in Fig.2 $weight(OP_{12}, OP_{42}) = 3$.

4.2 Semantics of Sequence Diagrams

The semantics of SDs is centered on the notion of events. First, we define the set of events associated with an SD in Subsection 4.2.1, and the different states of these events in Subsection 4.2.2. Then, we introduce the event-based semantics of SDs as LTS in Subsection 4.2.3.

4.2.1 Basic Definitions

Let EVT_G be the set of events associated with an SD, $E\_s = \{!m \mid m \in M\}$ and $E\_r = \{?m \mid m \in M\}$ be the set of sending events and the set of receiving events respectively. We consider $EVT = E\_s \cup E\_r$ a set of ordinary events. The fictitious events deal with the guard evaluation, and they ensure the synchronization between the lifelines. The fictitious events give us a high flexibility in defining execution strategies of the events, and defining a generic shape for the normal events of the SD. We consider Fic a set of the fictitious events. For each guarded operand, we define fictitious events. For each first event of the guarded operand X, we define a positive fictitious event τ_X^+ and a negative fictitious event τ_X^- associated with the guard evaluation True and False respectively. To simplify the formalization afterward, we assume that each non-empty guarded operand of a CF has only one first event. We consider $EVT_G = EVT \cup Fic$. In the following, we define some sets and functions that are used in the sequel.

- For a set of messages M we define two bijective functions. $F\_s : E\_s \rightarrowtail\!\!\!\!\!\rightarrow M$ that associates for each sending event one message, and $F\_r : E\_r \rightarrowtail\!\!\!\!\!\rightarrow M$ that associates for each receiving event one message.
- $F\_l : EVT_G \twoheadrightarrow L$ is a total surjective function that gives to each event one lifeline. For normal events, the associated lifeline is the sender or the receiver of the considered message. For fictitious events, the associated lifeline is the lifeline that executes the fictitious event, i.e., the lifeline of the first event of the considered operand.
- F_D gives the operand that directly contains an event: $F\_D : EVT_G \to OP$.
- F_G gives all the events that are contained in an operand including those contained in its nested operands:

$$\begin{cases} F\_G : OP \to \mathbb{P}(EVT), \\ F\_G(X) = F\_D^{-1}[((tree\_OP)^{+})^{-1}[\{X\}]]. \end{cases}$$

- The *sibling* binary relation relates two events, if they belong to distinct operands (directly or nested) of the same ALT, PAR or STRICT CF. For example in Fig.2, if the operands OP_{41} and OP_{42} belong to the same CF ALT, then they are siblings. The operands OP_{11} and OP_{12} belong to the same CF PAR, and then they are siblings.

We overload the relation $<_{\mathcal{SD},L}$ to support the partial order relation on each lifeline l_i between events. The relation $<_{\mathcal{SD},l}$ includes only the pairs of events that are directly consecutive with regard to the lifeline l; it is a minimal acyclic relation that contains no transitive implicates. By considering the transitive closure of the local order inside a lifeline $l <_{\mathcal{SD},l}^{+}$, we obtain all the visual precedence relation between events with regard to the lifeline l. We overload the function $weight$ that permits to associate with each event its maximal number of occurrence:

$$\begin{cases} weight : EVT_G \to NAT^{+}, \\ weight(e) = weight(OP_{00}, X), \\ \quad \text{where } X = F\_D(e). \end{cases}$$

4.2.2 Event States

According to a run, an event which belongs to a basic SD can have two obvious states: occurred or not yet occurred. However, these basic states are not enough to express the state of an event in an SD with complex structures (nested CF). Indeed, each event that belongs to such an SD can be: either not yet occurred, or occurred or ignored once or several times, or consumed. The variable $state$ is defined as follows: $state : EVT \to NAT$. The initial value of the state of each event is its weight that corresponds to its maximal value of occurrence in each run. The state of an event is decreased whenever it was 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 event e is executed or ignored once or several times: when $0 < state(e) < weight(e)$,

• consumed: when $state(e) = 0$.

4.2.3 Event-Based Semantics of Sequence Diagrams

Definition 5. *The semantics of the sequence diagram $\mathcal{SD}$ is defined as an LTS: $\langle EVT_G, S, s_0, s_F, C, <_{causG}, \rightarrow\rangle$, where:*

• *EVT_G is a set of events associated with $\mathcal{SD}$,*

• *S is a set of states,*

• *s_0 is the initial state, when all the events are* not yet occurred*,*

• *s_F is a final state, when all the events are* consumed*: $state = EVT_G \times \{0\}$,*

• *C is a set of constraints,*

• *$<_{causG} \subseteq EVT_G \leftrightarrow EVT_G$ denotes the causality relation between events computed from $\mathcal{SD}$, and*

• *$\rightarrow \subseteq S \times C \times EVT_G \times S$ is the transition relation corresponding to the occurrence of the events.*

The events represent incidents that cause SDs to transition from one state to another. The transition $(s, [g]evt, s')$ can be fired from state s if the following trigger conditions (TC) are checked: 1) the preceding events of the event evt are occurred (executed or ignored) (TC1), 2) the event evt is not yet consumed (TC2), and 3) the guard constraint $[g]$ is True (TC3). When transitions between states occur, some updating actions are performed as follows: 1) updating the state of the event evt (EE1), and 2) others variable updating required to deal with the synchronization issue (EE2). The computation of the causality relation $<_{causG}$ is presented in Subsection 4.3. The Event-based semantics of SDs expressed by LTS is modeled in Event-B (Section 5), where the formal definitions of the conditions TC1, TC2 and TC3 as well as the updating actions EE1 and EE2 are expanded in Subsection 5.2.

4.3 Computation of Causal Relations

In this subsection, we propose to generalize and to extend the causal relationship $<_{caus}$ firstly introduced in [26] for basic SDs, and extended in our previous work[7] dealing with SDs with some CFs (SEQ, ALT, LOOP, SEQ), to deal with new CFs (PAR and STRICT).

4.3.1 Global Causal Relation

The causal relation $<_{caus}$ is the union of the precedence relations (that we also call causal relations) between events of an SD. For a given event, its preceding events are the set of events whose execution enables its execution.

The causal relation $<_{caus}$ is redefined as:

$$<_{caus} = <_{Sync} \cup <_{RE} \cup <_{EE} \cup <_{RR} \cup <_{Hcaus} \cup <_{strict}.$$

The causal relations $<_{Sync}$, $<_{RE}$, $<_{EE}$ and $<_{RR}$ allow to compute the precedence relations for each event of an SD. The synchronization relationship ($<_{Sync}$) is unchangeable. The relationships $<_{EE}$, $<_{RE}$ (Subsection 4.3.2) and $<_{RR}$ (Subsection 4.3.3), are extended. We define the new relations $<_{Hcaus}$ (Subsection 4.3.4) and $<_{strict}$ (Subsection 4.3.5). In the next step, we introduce the relation $<_{\tau}$ induced by fictitious events. The relation $<_{\tau}$ permits to generate new precedence relations that must be considered with relationship $<_{caus}$. Then, the global causal order relation is defined as follows: $<_{causG} = <_{caus} \cup <_{\tau}$.

The relation $<_{causG}$ and all its subrelations are reflexive. The relation $<_{causG}$ expresses the causal dependency between events. It includes preceding events for each event: if we have $e <_{causG} e'$, e' is an immediate preceding event of e; the transitive closure of the global relation $<^{+}_{causG}$ provides all preceding events.

Note that some events in the relation $<_{causG}$ are in conflict with each other (i.e., we have $e_1 <_{causG} e'$ and $e_2 <_{causG} e'$ where e_1 and e_2 are located in distinct operands of the same ALT CF; hence they cannot be executed in the same trace). We will see later that this constraint will be ensured by the synchronization. Indeed, for an event to be enabled, its preceding events must be either executed or disabled.

The relation $<_{causG} - <_{Hcaus}$ is an acyclic relation. Indeed, the relation $<_{Hcaus}$ can contain a cycle dependency between events, because in the relation $<_{causG}$ we do not distinguish between events from different iterations (of a LOOP CF) associated with the same sending (resp. receiving) message. This will be considered, as we will see later by using a counter variable (the weight of the event) that decrements in each iteration.

We consider the function $first$ that gives the first event of each operand: $first : OP \rightarrow EVT$. A first event is an event that has no preceding events in the considered operand:

$$\begin{aligned} &first \\ &= \{(X, e') | X \in OP \wedge e \in EVT \wedge e' \in F\_G(X) \wedge \\ &\quad (\forall e)[e \in EVT \wedge e \in F\_G(X) \wedge (e, e') \notin <^{+}_{caus}]\}. \end{aligned}$$

4.3.2 Causal Relations $<_{EE}$ and $<_{RE}$

To alleviate the presentation of the formalization of the relations $<_{EE}$ and $<_{RE}$, we introduce the relation $succ$ that relates two events that belong to the

same lifeline and they are successive. We admit between them, events that belong to an operand that can be omitted (i.e., the events between successive events do not belong to any ancestor operand of the operand of the considered event), and receiving events.

The relation *succ* is defined as:

$$\begin{aligned} & succ \\ = \{(e,e')|(e,e') \in EVT^2 \wedge \\ & (\exists l)[l \in L \wedge (e <^{+}_{\mathcal{SD},l} e') \wedge \\ & (\forall e'')[(e <^{+}_{\mathcal{SD},l} e'') \wedge (e'' <^{+}_{\mathcal{SD},l} e') \Rightarrow e'' \in E_r \vee \\ & F\_D(e'') \notin (anc(F\_D(e)) \cup anc(F\_D(e')))]]\}. \end{aligned}$$

The relationship $<_{EE}$ permits to order two sending events, if they are successive and they are not siblings operands:

$$\begin{aligned} <_{EE} = \{(e,e')|(e,e') \in E\_s^2 \wedge \\ (e,e') \notin sibling \wedge (e,e') \in succ\}. \end{aligned}$$

The relationship $<_{RE}$ permits to order two events such that the first one is a receiving event and the second one is a sending event, and both of them satisfy the conditions expressed in not *sibling* and *succ* relations. The relationship $<_{RE}$ is defined as:

$$\begin{aligned} <_{RE} = \{(e,e')|e \in E\_r \wedge e' \in E\_s \wedge \\ (e,e') \notin sibling \wedge (e,e') \in succ\}. \end{aligned}$$

4.3.3 Causal Relation $<_{RR}$

If the communication between distributed components ensures a first in first out delivery order, then two messages coming from the same lifeline to the same target lifeline are received in the same order of their emission. The relationship $<_{RR}$ permits to compute this precedence relation:

$$\begin{aligned} & <_{RR} \\ = \{(e,e')|(e,e') \in E\_r^2 \wedge F\_l(e) = F\_l(e') \wedge \\ & (\exists e_1)(\exists e_1')[(e_1,e_1') \in E\_s^2 \wedge (e_1 <^{+}_{EE} e_1') \wedge \\ & F\_r(e) = F\_s(e_1) \wedge F\_r(e') = F\_s(e_1')]\}. \end{aligned}$$

4.3.4 Causal Relation $<_{Hcaus}$

The identification of precedence relationships in a LOOP CF is rather fastidious. Indeed the events inside a LOOP CF, from the second iteration, can have preceding events that can be located in the same LOOP CF of the previous iterations. We call hidden precedence relations $<_{Hcaus}$, the relations between the events of LOOP CF of the current iteration and the events of the previous iterations. In order to compute the hidden precedence relations LOOP operand named X, we propose the following steps (see the example in Fig.6): we flatten the LOOP CF only once; 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. We define the set EVT' to represent the events of the next iteration and we compute the causal relations $<'_{RE}$, $<'_{EE}$ and $<'_{RR}$ between EVT and EVT'. We denote by $<_{HcausX}$, the hidden precedence relations of a given LOOP CF named

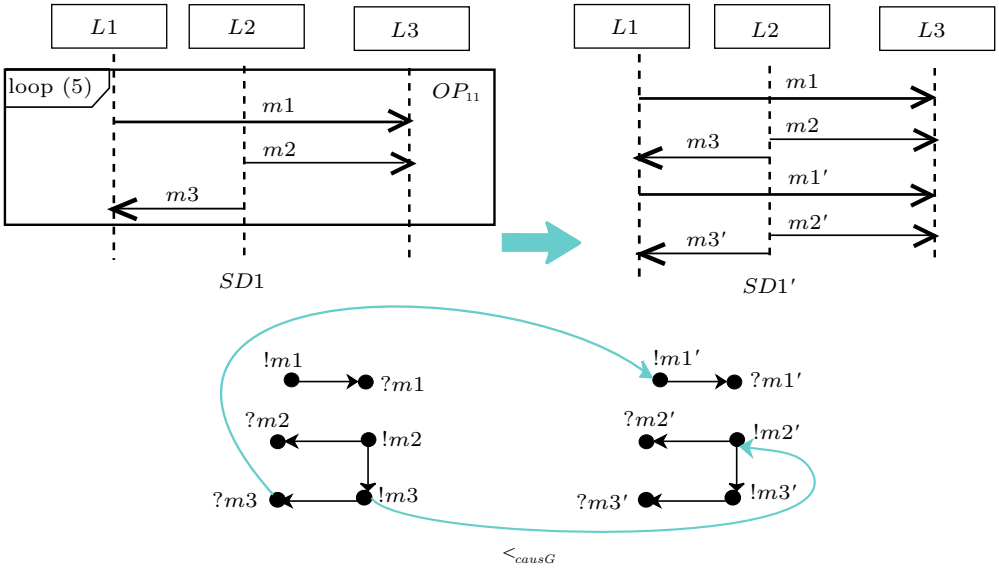

Fig.6. Processing of an SD with a LOOP CF.

X and we give the following formalization.

$$<_{HcausX} = \{(e,e')|(e,e') \in (EVT \times EVT') \wedge (e,e') \in (<'_{RE} \cup <'_{EE} \cup <'_{RR})\}.$$

In the case where we have several LOOP CFs that can be sequenced or nested, we apply the same processing by computing for each LOOP CF its precedence relations; the entire hidden precedence relation is the union of the hidden precedence relations of each LOOP CF. We present in Fig.6 the sequence diagram $SD1$, the sequence diagram $SD1'$ that represents the flattening of the LOOP CF and the causal order relation associated with $SD1$. For each iteration the order relation between events {(!m1,?m1), (!m2,?m2) (!m3,?m3) (!m2,!m3)} must be respected and between the iterations we compute this order relation $<_{HcausOP_{11}}$= $\{(?m3, !m1'), (!m3, !m2')\}$.

4.3.5 Causal Relation $<_{strict}$

There is no interleaving of the occurrence of the events that belong to different operands of a strict CF. The events of an intermediate operand can only occur when all the events of the previous operand occurred. There is no interleaving of the occurrence of the events that belong to different operands of a strict CF. The events of an intermediate operand can only occur when all the events of the previous operand occurred. This causality relationship is defined by the relation named $<_{strict}$: for each STRICT CF named CF_i with a set of operands $OP_i = \{OP_{i1}, ..., OP_{ik}\}$, we compute:

$$<_{strict_i} = \{(e,e')|(\forall j)[j \in [2,k] \wedge e \in F\_G(OP_{ij-1}) \wedge e' = first(OP_{ij})]\}.$$

Then $<_{strict}$ is the union of all computed $<_{strict_i}$.

4.3.6 Relations for Fictitious Events

The insertion of fictitious events in the SD leads to new precedence relations. In a guarded operand, the two associated fictitious events have the same preceding events and are inserted between the first event and the preceding events of the first event. In a nested CF, we assume that the guard evaluation of a child operand should be made after that the evaluation of the parent operand guard is True. These precedence relations are computed from a relation that we denote $<_\tau$. The relation $<_\tau$ is defined as the union of the sub-relations $<_{\tau 1}$, $<_{\tau 2}$ and $<_{\tau 3}$ that are detailed in the following.

For an operand X, the fictitious events τ_X^+ and τ_X^- are located between the first event of the considered operand and all its preceding events. Then, it inherits all the preceding events of the first event of OP_{ij} operand. These new precedence relations are computed in the relation $<_{\tau 1}$:

$$<_{\tau 1} = \{(e,e')|e \in EVT \wedge e' \in Fic \wedge (e <_{caus} first(F\_D(e')))\}.$$

The fictitious events τ_X^+ and τ_X^- become preceding events of the first event of the operand X; this is computed from $<_{\tau 2}$ relation:

$$<_{\tau 2} = \{(e,e')|e \in Fic \wedge e' \in EVT \wedge e' = first(F\_D(e))\}.$$

The fictitious events of the nested operand should be run after the execution of the fictitious events of the parent operand. This causality relationship is defined by the relation named $<_{\tau 3}$:

$$<_{\tau 3} = \{(e,e')|e \in Fic^2 \wedge first(F\_D(e)) = first(F\_D(e')) \wedge tree\_OP(F\_D(e')) = F\_D(e)\}.$$

From now, in Section 5 we define how the formal definition of SDs together with their interpretation semantics is encoded within Event-B.

5 Translation of SDs into Event-B

In this section, we translate UML 2.X sequence diagrams into Event-B specifications. The formal definition of an SD (Subsection 4.1) is captured in the contexts (Subsection 5.1); the behavior of an SD (Subsection 4.2) is captured in a machine (Subsection 5.2) by means of variables whose values are modified by events. Hence, we propose a generic architecture of the translation of any SD with nested CFs into Event-B specification, as depicted in Fig.7. For each SD, we define three contexts and a machine. In the sequel, we detail the contents of each component of the architecture.

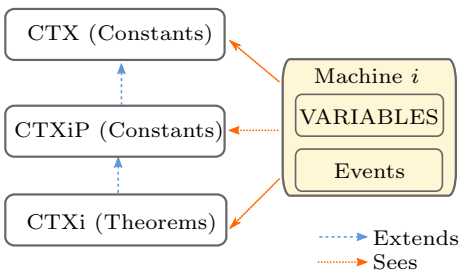

Fig.7. Generic architecture of the translation.

5.1 Construction of the Contexts

Consider the following sequence diagram:

$$\mathcal{SD} = \langle L, M, <_{\mathcal{SD},L}, OP, F, tree\_OP \rangle,$$

and its associated LTS

$$\langle EVT_G, S, s_0, s_F, C, <_{causG}, \rightarrow \rangle.$$

We translate the formal definition of the SD into contexts, in which we define the sets, the constants, the axioms, as well as the precedence relations. Indeed, in the first and the second context (CTX, $CTXiP$), we express the concrete values of the components of the considered SD (lifelines, messages, events, operands, etc.) that are defined by sets and constants. An example of the context $CTXiP$ is given in Fig. 8. The contents of the context $CTXi$ is unchanged for any encoded SD; this context contains the typing of constants and their properties. The context $CTXi$ is detailed in Subsection 6.1. The context $CTXiP$ extends the context CTX and it is extended by the context $CTXi$.

The sets L, OP, M and EVT are translated as carrier sets in CTX by $LIFELINES$, $OPERANDS$, MSG and EVT respectively. Table 4 illustrates the translation of each component of the SD formal definition, as well as the precedence relations, into constant relations in $CTXiP$.

CONTEXT ctx1P
EXTENDS ctx
AXIOMS

axm1: $CausSync = \{(e\_connect1 \mapsto r\_connect1),(e\_connect2 \mapsto r\_connect2),(e\_view\_list \mapsto r\_view\_list),$
$(e\_list\_queries \mapsto r\_list\_queries),(e\_return\_list \mapsto r\_return\_list),(e\_display\_list \mapsto r\_display\_list),$
$(e\_input\_query \mapsto r\_input\_query),(e\_seek\_query \mapsto r\_seek\_query),(e\_result\_query \mapsto r\_result\_query),$
$(e\_transmit\_result \mapsto r\_transmit\_result),(e\_not\_found \mapsto r\_not\_found),(e\_display\_not\_found \mapsto r\_display\_not\_found)\}$

axm2: $CausEE = \{(e\_connect1 \mapsto e\_view\_list),(e\_connect2 \mapsto e\_input\_query)\}$

axm3: $CausRE \in A\_EVT \leftrightarrow A\_EVT$

axm4: $CausRE = \{(r\_connect1 \mapsto e\_list\_queries),(r\_view\_list \mapsto e\_list\_queries),(r\_list\_queries \mapsto e\_return\_list),$
$(r\_return\_list \mapsto e\_display\_list),(r\_connect2 \mapsto e\_seek\_query),(r\_input\_query \mapsto e\_seek\_query),$
$(r\_seek\_query \mapsto e\_result\_query),(r\_result\_query \mapsto e\_transmit\_result),$
$(r\_seek\_query \mapsto e\_not\_found),(r\_not\_found \mapsto e\_display\_not\_found)\}$

axm5: $HCaus = \{(e\_input\_query \mapsto e\_input\_query),(e\_transmit\_result \mapsto e\_seek\_query),(e\_seek\_query \mapsto e\_seek\_query),$
$(e\_display\_not\_found \mapsto e\_seek\_query)\}$

axm6: $Caus = CausSYNC \cup CausEE \cup CausRE \cup HCaus$

axm7: $CausT1EE = \{(e\_connect2 \mapsto T\_op11pos),(e\_connect2 \mapsto T\_op11neg)\}$

axm8: $CausT1RE = \{(r\_seek\_query \mapsto T\_op31pos),(r\_seek\_query \mapsto T\_op31neg),(r\_input\_query \mapsto T\_op21pos),$
$(r\_input\_query \mapsto T\_op21neg),(r\_connect2 \mapsto T\_op21pos),(r\_connect2 \mapsto T\_op21neg),(r\_seek\_query \mapsto T\_op32pos),$
$(r\_seek\_query \mapsto T\_op32neg)\}$

axm9: $HCausT1 = \{(e\_input\_query \mapsto T\_op11pos),(e\_input\_query \mapsto T\_op11neg),(e\_display\_not\_found \mapsto T\_op21pos),$
$(e\_display\_not\_found \mapsto T\_op21neg),(e\_transmit\_result \mapsto T\_op21pos),(e\_transmit\_result \mapsto T\_op21neg),$
$(e\_seek\_query \mapsto T\_op21pos),(e\_seek\_query \mapsto T\_op21neg)\}$

axm10: $CausT1 = CausT1EE \cup CausT1RE \cup HCausT1$

axm11: $CausT2 = \{(T\_op11pos \mapsto e\_input\_query),(T\_op21pos \mapsto e\_seek\_query),$
$(T\_op31pos \mapsto e\_result\_query),(T\_op32pos \mapsto e\_not\_found)\}$

axm12: $EVT\_FIRST = \{e\_input\_query, e\_seek\_query, e\_result\_query, e\_not\_found\}$

axm13: $CausEEG = (CausEE \rhd EVT\_FIRST) \cup CausT1EE$

axm14: $CausREG = (CausRE \rhd EVT\_FIRST) \cup CausT1RE$

axm15: $HCausG = (HCaus \rhd EVT\_FIRST) \cup HCausT1$

axm16: $CausG = CausSYNC \cup CausEEG \cup CausREG \cup HCausG \cup CausT2 \cup CausT3$

axm17: $Begin = \{e\_connect1\}$

axm18: $A\_OPERANDS = \{OP00, OP11, OP41, OP42, OP21, OP31, OP32\}$

axm19: $A\_WEIGHT = \{(e\_connect1 \mapsto 1),(e\_connect2 \mapsto 1),(e\_view\_list \mapsto 1),(e\_list\_queries \mapsto 1),(e\_return\_list \mapsto 1),$
$(e\_display\_list \mapsto 1),(e\_input\_query \mapsto 2),(e\_seek\_query \mapsto 3),(e\_result\_query \mapsto 3),(r\_connect1 \mapsto 1),(r\_connect2 \mapsto 1),$
$(r\_view\_list \mapsto 1),(e\_transmit\_result \mapsto 3),(e\_not\_found \mapsto 3),(e\_display\_not\_found \mapsto 3),(r\_input\_query \mapsto 2),$
$(r\_seek\_query \mapsto 3),(r\_result\_query \mapsto 3),(r\_transmit\_result \mapsto 3),(r\_not\_found \mapsto 3),(r\_display\_not\_found \mapsto 3),$
$(T\_op11pos \mapsto 2),(T\_op11neg \mapsto 2),(T\_op21pos \mapsto 3),(T\_op21neg \mapsto 3),(T\_op31pos \mapsto 3),(T\_op31neg \mapsto 3),(T\_op32pos \mapsto 3)$
$(T\_op32neg \mapsto 3),(r\_list\_queries \mapsto 1),(r\_return\_list \mapsto 1),(r\_display\_list \mapsto 1)\}$

axm20: $EVT\_OP41 = \{e\_connect1, e\_view\_list, e\_list\_queries, e\_return\_list, e\_display\_list, r\_connect1, r\_view\_list, r\_list\_queries,$
$r\_return\_list, r\_display\_list\}$

axm21: $EVT\_OP42 = \{e\_connect2, r\_connect2\}$

axm22: $EVT\_OP11 = \{e\_input\_query, r\_input\_query, T\_op11pos, T\_op11neg\}$

axm23: $EVT\_OP21 = \{e\_seek\_query, r\_seek\_query, T\_op21pos, T\_op21neg\}$

axm24: $EVT\_OP31 = \{e\_result\_query, r\_result\_query, e\_transmit\_result, r\_transmit\_result, T\_op31pos, T\_op31neg\}$

axm25: $EVT\_OP32 = \{e\_not\_found, r\_not\_found, e\_display\_not\_found, r\_display\_not\_found, T\_op32pos, T\_op32neg\}$

axm26: $EVT\_OP00 \subseteq A\_EVT$

axm27: $EVT\_OP00 = \varnothing$

END

Fig.8. Context $CTX1P$ with some axioms of the specification.

Table 4. Mapping Between Elements of an SD and Event-B Encoding

SD	Event-B
$<_{SyncG}$	$causyncG$
FCT_r	FCT_EVT_R
$<_{Sync}$	$causync$
$<_{EE}$	$causEE$
$<_{causG}$	$causG$
$<_{EEG}$	$causEEG$
$<_{HcausG}$	$HcausG$
$<_{REG}$	$causREG$
FCT_s	FCT_EVT_S
E_r	EVT_R
E_s	EVT_S
FCT_l	FCT_l
$<_{RE}$	$causRE$
$<_{Hcaus}$	$Hcaus$
$<_{\tau 1}$	$causT1$
$<_{\tau 2}$	$causT2$
$<_{\tau 3}$	$causT3$

Fig. 8 depicts the implementation of the context $CTX1P$ associated with the SD in Fig.2. In the context $CTX1P$, the axioms $axm10..axm15$ express the computation of the precedence relations $<_{Sync}$, $<_{RE}$, $<_{EE}$, and hidden precedence relations $<_{Hcaus}$. To facilitate the computation of some precedence relations (Fig.8), we define new sets: for instance, in axiom 10, to compute relation $causT1$, we define in the axioms 7–9 two new sets $causT1EE$, $causT1RE$ and $HCausT1$. In the axiom 13, we define the new set EVT_FIRST that is used for the computation of $causEEG$, $causREG$, $HcausG$ and $causG$ precedence relations. In the axiom 19, we associate with each event its maximal number of occurrence (its weight). In the axioms $axm20..axm27$, we associate with each operand its events.

5.2 Construction of the Event-B Machine

The Event-B machine has a visibility on the contexts. It encapsulates state variables, invariant on them and its initialization, and guarded events that describe how and when the state should be updated. The initialization is a special event, with a true guard, and it sets the initial state of the machine. An event is made of guards that denote its enabling conditions, and actions that denote the way the state is modified by the event. An event occurrence is supposed to take no time. Hence, no two events can occur simultaneously. When we have several enabled events (with true guards), one of them is chosen non-deterministically to occur, and its actions modify the state.

We construct for the sequence diagram $\mathcal{SD}$ an Event-B machine. We define the variables *state* and $current\_lifeline$ that express the state of each event and the lifeline that executes the current event respectively. The events of the Event-B machine are composed by the normal events of the SD and the fictitious events. Each event occurs under some trigger conditions and produces some execution effects. Let *evt* be an event that belongs to the Y operand ($Y = F\_D(e)$): $evt \mathrel{\widehat{=}} TC|EE$, where the guard TC is a conjunction of the trigger conditions TC_i and the action EE is the union of the execution effects EE_i. In the next two subsections (Subsection 5.2.1 and Subsection 5.2.2), we will detail the construction of the Event-B event *evt*.

5.2.1 Guards of Event-B Events

The guards are the trigger conditions of an event that consist in checking the following conditions. The preceding events of the current event are occurred (executed or ignored) and (TC1, TC2 and TC3) is the first trigger condition that is essentially based on the comparison of the state of the current event and the states of its preceding events. We consider the event *evt* and its preceding event e, and three cases should be explored: TC1, TC2 and TC3.

• The considered events are located in the same CF, and then they have the same initial weight. We define TC1 as follows:

$$\mathsf{TC1} : state(e) < state(evt).$$

• The considered events are located in different CFs. The comparison of the states of two events is based on their weights by relative to a first common ancestor (LCA). Let X, Y and Z be three operands such that $X = F\_D(e)$, $Y = F\_D(evt)$ and $Z = LCA(X, Y)$. The generic formula of the first trigger condition is:

$$\begin{aligned}\mathsf{TC2} : &((state(e) \bmod weight(Z, X) = 0)) \wedge \\ &(state(e)/weight(Z, X) < \\ &state(evt)/weight(Z, Y)).\end{aligned}$$

• For each event *evt* of a LOOP CF that has hidden precedence relations, we define the trigger condition TC3 to check that the preceding events that are computed from the hidden precedence relation have occurred. We consider the LOOP CF Z and $e <_{HcausZ} evt$:

$$\begin{aligned}\mathsf{TC3} : &(second = \text{true}) \Rightarrow \\ &((state(e) \bmod weight(Z, X) = 0)) \wedge \\ &(state(e)/weight(Z, X) = \\ &state(evt)/weight(Z, Y)),\end{aligned}$$

where *second* is true when the event *evt* is from the second iteration of the LOOP CF Z. In order to illustrate the trigger conditions TC1, TC2 and TC3, we consider the SD and its associated tree in Fig.9. We detail the following examples.

• The events $!m1$ and $!m2$ are in the same operand and $(!m1, !m2) \in <_{EE}$. In the guard of the event $!m2$, we check that: $state(!m1) < state(!m2)$ (TC1).

• The events $!m2$ and $!m3$ are in different operands and $(!m2, !m3) \in <_{EE}$. In the guard of event $!m3$, we check that: $state(!m2)\ mod\ 3 = 0 \wedge state(!m2)/3 < state(m3)/2$ (TC2).

• We have $(!m3, !m1) \in <_{HcausOP_{21}}$. In the guard of the event $!m1$, we check that: $state(!m3)\ mod\ 2 = 0 \wedge state(!m3)/2 = state(!m1)/3$ (TC3).

The second condition is that the current event *evt* is not yet consumed (TC4). TC4 is expressed as follows:

$$\mathsf{TC4} : state(evt) \geqslant 1.$$

The fictitious events have, in addition to TC1, TC2, TC3 and TC4, an additional trigger condition TC4 in which we check the value of guard of the considered operand.

5.2.2 *Execution Effects of the Event-B Events*

Each action of an event of an Event-B machine can be composed by one or several execution effects (EE1 and EE2). For the normal events, the execution effects consist in:

1) updating the state of the event (EE1):

$$\mathsf{EE1} : state(evt) = state(evt) - 1, \text{ and}$$

2) updating the lifeline of the current event (EE2):

$$\mathsf{EE2} : current\_lifeline = F\_l(evt).$$

The fictitious events have, in addition to EE1 and EE2, other execution effects that are required to deal with the synchronization issue. For an ALT and a guarded LOOP CF, an important issue should be handled after the guard evaluation, and it is the synchronization between the lifelines covered by these CFs. Indeed, the standard semantics of an ALT CF imposes that only one operand must be chosen among several potential operands. When none guard is true, the CF is omitted. Moreover, for the guarded LOOP operand, the events inside the operand can occur while the guard is evaluated to true and the maximal number of iterations is not reached. We consider the event *evt* as a fictitious event, and it should produce execution effects EE3, EE4 and EE5, which allow the synchronization. Three cases should be explored.

• Updating the state of its dual fictitious event evt' (negative or positive) of the same operand:

$$\mathsf{EE3} : state(evt') = state(evt') - 1.$$

• Only for execution effect of a positive fictitious event of an operand of an ALT CF, by decrementing the state of each event of the siblings operands of the Y operand ($Y = F\_D(evt)$):

$$\mathsf{EE4} : (\forall e)(\forall B)[B \in sibling(Y) \wedge e \in F\_G(B)) \Rightarrow (state(e) = state(e) - weight(B, F\_D(e)))].$$

• Only for the execution effect of a negative fictitious event of an operand of ALT and LOOP CFs, decrementing (with respect to its weight relative to the operand) the state of each event of the considered operand to

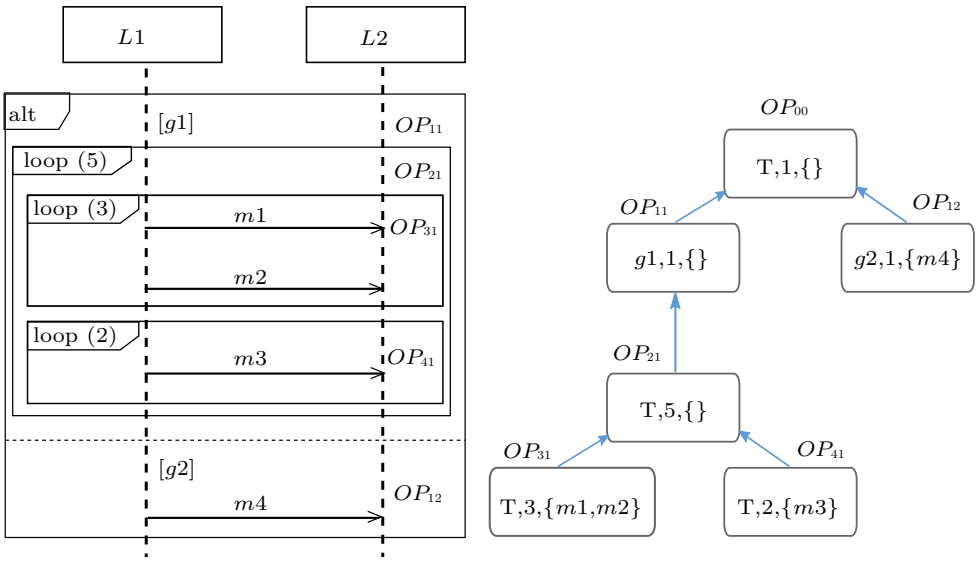

Fig.9. Illustration of the weight of the path in the tree. T: true.

prohibit their occurrence. A negative fictitious event of a LOOP operand should decrement the state of each event of the considered operand with multiplication by the number of the remaining iterations that must be ignored:

$$\mathsf{EE5}: (\forall e)[e \in F\_G(X) \Rightarrow state(e) = state(e) - (w \times weight(Y, F\_D(e)))],$$

where w is the number of the remaining iterations for the LOOP CF and $w = 1$ for the ALT CF.

5.3 Machine of Motivating Example

Fig. 10 depicts the skeleton of the machine of the motivating example (Fig.2). The encoding of Event-B specifications can be found in the technical report [28]. The typing of the variables is defined in the invariant clause. In the initialization clause, we initialize the state of each event with its weight that corresponds to its maximal number of occurrences in each run. The variable *current_lifeline* is initialized with the lifeline of the event authorized to occur.

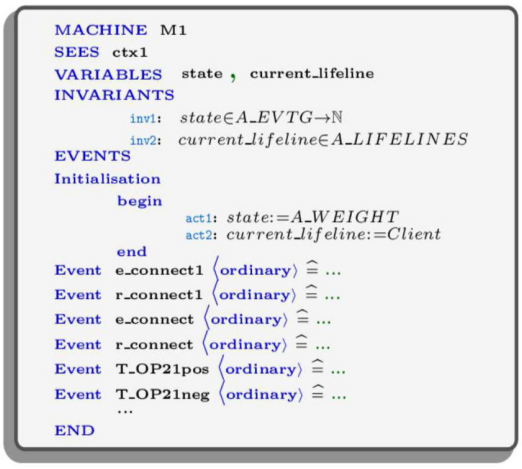

Fig.10. Skeleton of the machine of the motivating example.

5.3.1 Example of Normal Event Implementation

Fig. 11 depicts the implementation of the normal event defined by $(!connect1, Server)$. The guard grd1 checks that the preceding events that are located in the OP_{11} operand were occurred; the guard grd2 checks that the event can still occur. After its occurrence, we update the value of its state and the variable $current\_lifeline$ in the first and the second actions, respectively.

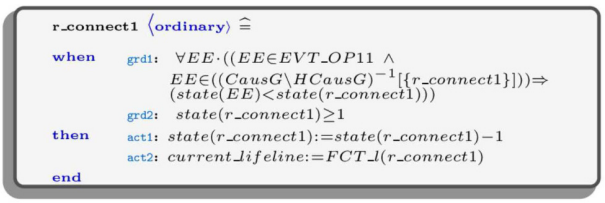

Fig.11. Implementation of event $r\_connect1$.

5.3.2 Examples of Fictitious Event Implementation

In Fig.12, we show the implementation of the positive fictitious event of the OP_{21} of the LOOP CF.

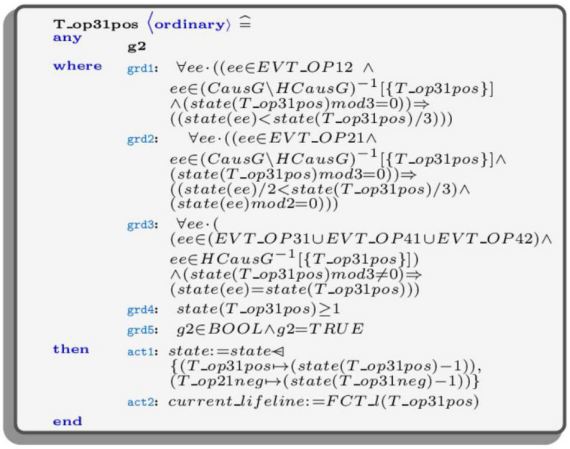

Fig.12. Implementation of the event $T\_OP31pos$.

The guards grd1 and grd2 check the states of the preceding events that are located in the OP_{12} and in OP_{21} operands. The guard grd3 checks the states of preceding events that are located in the operands OP_{31}, OP_{41} and OP_{42}. These preceding events are defined in a hidden precedence relation $HcausG^{-1}$. The guards grd4 and grd5 check that the event can still occur and the guard $[g2]$ is true respectively. The actions produced by the positive fictitious event $(T\_OP31pos)$ allow to update the value of its state, the value of the state of the negative fictitious event $(T\_OP31neg)$ and the value of the variable $current\_lifeline$.

Fig. 13 represents the implementation of the negative fictitious event $(T\_OP41neg)$ of the OP_{41} operand (ALT CF). The guard grd1 checks that the preceding events, located in OP_{31} operand, were occurred; the guard grd2 checks that the event can still occur; and the guard grd2 checks that the guard $[g3]$ is evaluated to false. The first action act1 decrements the states of each event of the considered operand OP_{41} to prohibit their occurrences.

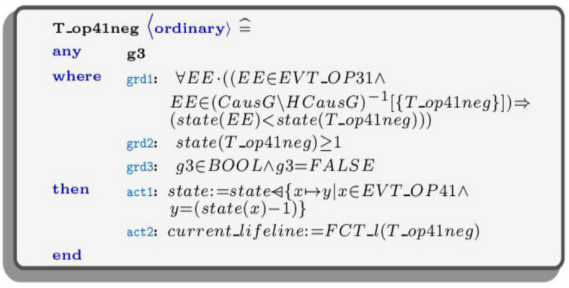

Fig.13. Implementation of the fictitious event $T\_OP41neg$.

In Section 6, we present how the transformation of UML 2.X SD into Event-B allows us to perform several kinds of verification including simulation, trace acceptance, verification of properties, and verification of refinement relation between SDs.

6 Analysis and Verification of Sequence Diagrams

In this section, we detail different applications of our transformational semantics of the UML 2.X SDs.

6.1 Well-Formed Sequence Diagrams

We check the formal specification of SD in Event-B, by considering the well-formedness properties, using the context $CTXi$. Technically this involves checking automatically with ProB that the instantiation of the context $CTXi$ is correct. In the context $CTXi$ (Fig.14), we define the axioms; they allow to specify the typing of constants or to express properties on them. Thus, they are marked as theorems. The proofs of $CTXi$ allow us to verify that the considered SD is well-formed and consistent. The content of this context is generic. For instance, in the axioms 1..14, we express that the SD is not empty, in the axiom 26, we check that each event belongs to one lifeline; in the axioms 16 and 17, we express that the causal relations $<_{caus}$ and $<_{causG}$ are non-reflexive and they are acyclic in the axioms 20 and 23.

6.2 Trace Acceptance and Simulation

ProB [12] offers the possibility to check traces by proposing in each step of the execution the list of the enabled events. Indeed, ProB offers a manually driven and checked animation that allow to analyze, to check and to validate some expected behaviors of the modeled system. Fig.15 presents the animation of the Event-B specification of the motivating example. The transition diagram of the state space generated with the ProB animator for the motivating example contains 164 818 checked states and 534 909 transitions.

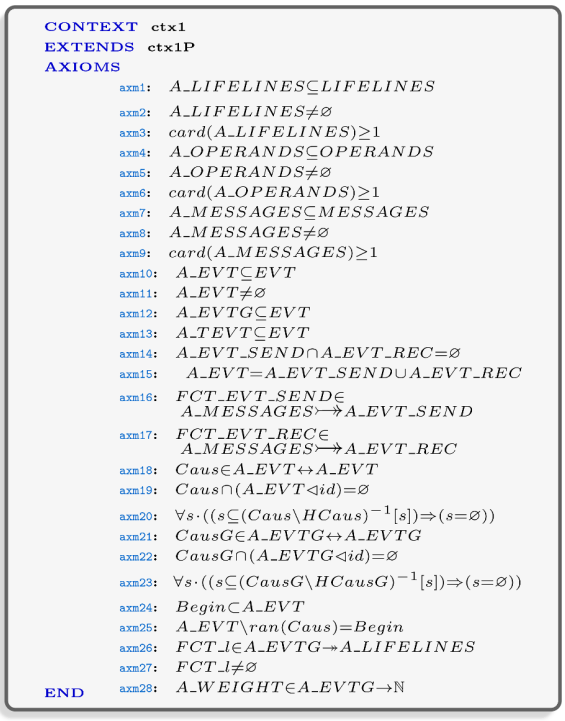

Fig.14. Theorems of the context $CTX1$.

6.3 Checking of Temporal Properties

ProB provides support for LTL (linear temporal logic) model checking. Indeed, ProB supports an extended version of LTL, LTL$^{[e]}$. In contrast to the standard LTL that only supports states, LTL$^{[e]}$ provides support for propositions on transitions. In practice, writing propositions on transitions is allowed by using the constructs $e(...)$ and $e[...]$. The operators G and F denote globally and finally respectively; they are temporal operators that express the future; the operator O denotes once, and it is a temporal operator that expresses the past (it is the dual of F operator). A machine M satisfies a property P if all traces of M satisfy P. We can express some properties that are intrinsic to the studied system, as we can express general properties that can be checked for any specification.

In our motivating example, we propose the checking of some LTL properties. Fig.16 is a screenshot of some properties that we have expressed in ProB.

For example, we check these two properties.

- $P1$ ($Fig.17$). We can express a property that permits to check that each occurrence of the event

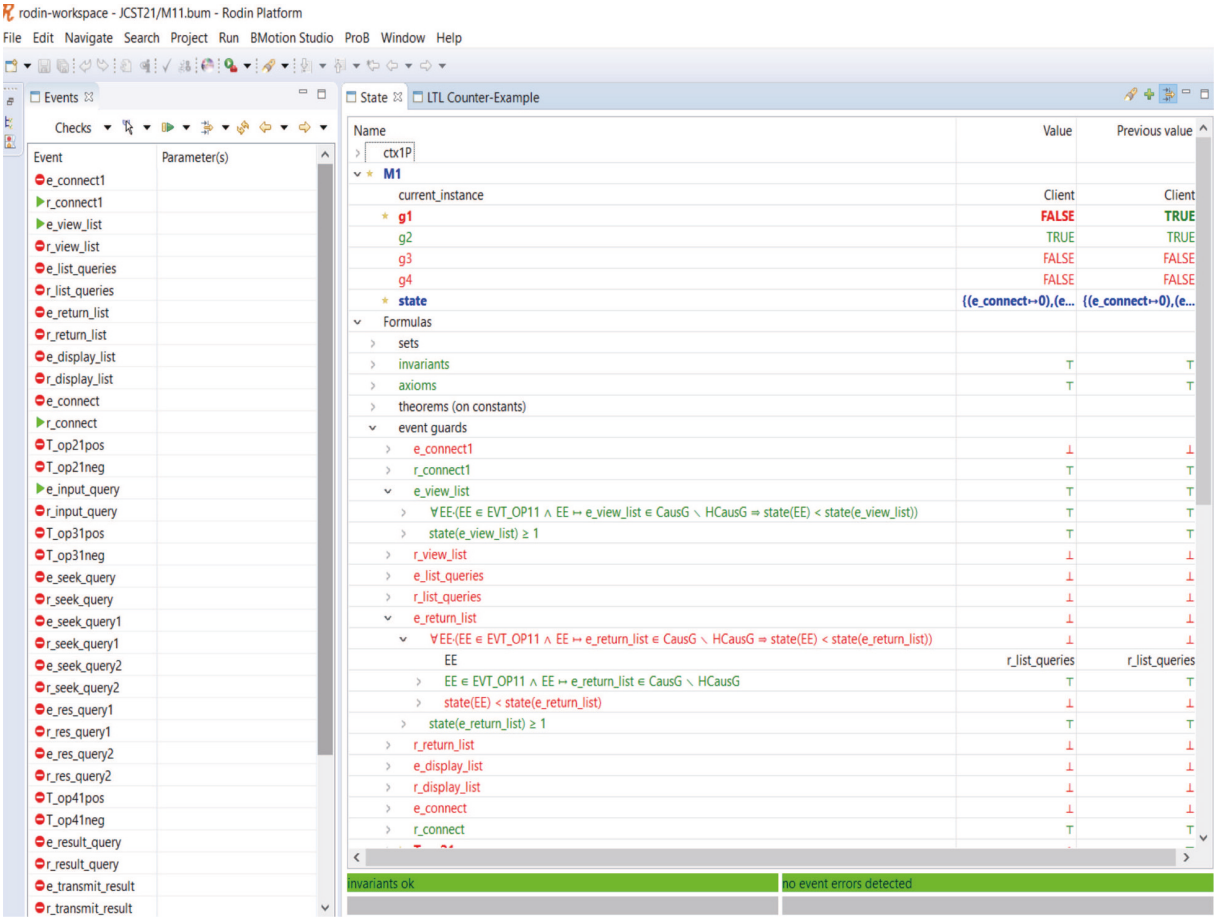

Fig.15. Screenshot of the animation of the Event-B specification of the motivating example.

r_seek_query will be eventually followed by either the event $r\_transmit\_result$ or the event $r\_display\_not\_found$.

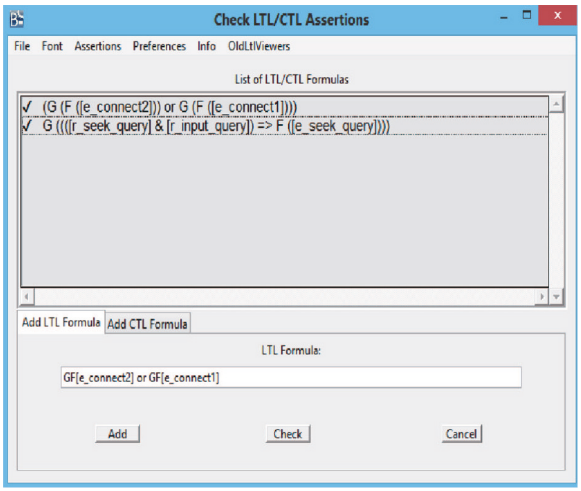

Fig.16. LTL/CTL assertions viewer of ProB.

$$P1\colon G([r\_seek\_query] \Rightarrow F([r\_transmit\_result] \vee [r\_display\_not\_found]))$$

Fig.17. LTL$^{[e]}$ formula of $P1$ property.

• $P2$ ($Fig.18$). Each sent message will eventually be received.

$$\begin{aligned} P2\colon\ & G([e\_seek\_query] \Rightarrow F([r\_seek\_query])) \ \&\\ & G([e\_connect2] \Rightarrow F(e[r\_connect2])) \ \&\\ & G\ ([e\_input\_query] \Rightarrow F([r\_input\_query])) \ \&\\ & G([e\_result\_query] \Rightarrow F([r\_result\_query])) \ \&\\ & G([e\_transmit\_result] \Rightarrow F([r\_transmit\_result])) \ \&\\ & G([e\_not\_found] \Rightarrow F([r\_not\_found])) \ \&\\ & G([e\_display\_not\_found] \Rightarrow F([r\_display\_not\_found])) \end{aligned}$$

Fig.18. LTL$^{[e]}$ formula of $P2$ property.

For a given formula, three status are possible: 1) it

has not been checked yet; 2) it is true for all valid paths; 3) it is false; in this case the model checker searches for a counterexample (i.e. a path that does not satisfy the current formula).

Fig.19 represents a partial view of the counter example reported by the LTL model checking for the formula of the $P3$ property. The counter example is a finite path leading to a deadlock state (the colored state in red). This result is logical since if all the guards of the operands of the ALT CF are false we do not obtain after the occurrence of the event r_seek_query neither the event $r\_transmit\_result$ nor the event $r\_display\_not\_found$.

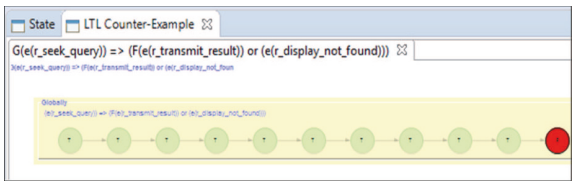

Fig.19. Screen shot of the LTL viewer.

7 Conclusions

To help in preliminaries design steps of distributed systems, we proposed interpretation semantics in which the meaning of SDs with nested CFs (SEQ, ALT, OPT, LOOP, STRICT and PAR) is unambiguously interpreted using the partial order relations on events that describe the behavior of SDs. Moreover, in our semantics we proposed a new formalization of SDs with nested CFs based on a tree structure; this formalization is user-friendly since it permits the determination of the relationships between the operands of the SD with some functions and relations we defined. The use of a tree structure permits a straightforward identification of the precedence relations of each event in an SD with nested CFs. The generation of traces depends on partial order. To compute the partial order between the events, the formalized rules are directly relevant without the need neither to make a preprocessing on SD nor to add operators or messages or global controller to get the desired order. Hence, we facilitated the task for software practitioners. Then, our semantics includes an approach for the guard evaluation and synchronization that takes into account the independence of the distributed components. The proposed semantics can be enhanced to consider other important aspects like time constraints and to handle specific properties. We aimed to cover the conformance operators, as well as the gate feature that cause the ill-formedness problem. Finally, we implemented the formal description of SDs with their causal semantics with Event-B by preserving the essential features of SDs and benefited from the several advantages of this framework. This transformational semantics enables rigorous model analysis, the checking of the correctness of the model and some properties using the formal techniques of Event-B and its tools supports: Rodin and ProB model-checker.

Toward the Checking of Refinement Relation. We currently studied an important concept that is the refinement of SDs corresponding to an iterative and incremental development processes mainly used to handle complex systems. SDs are used to capture the requirements of a system, and stepwise refinement creates a relationship between the requirements and the implementation. There are two possibilities to refine a sequence diagram. Intra-level refinement is applied to sequence diagrams that belong to the same level of abstraction, and it is used to reveal the internal behavior of a component. Inter-level refinement is used to transfer a sequence diagram to the next lower level of abstraction by substituting the lifelines with several more detailed lifelines. Obviously, new messages and events can be added in the refined SD. CF and guard (for instance, by strengthening it to reduce the non-determinism) can be also refined. The refinement of an SD that we defined relates to its structure (its components), and its semantics (trace refinement).

Refinement in Event-B allows to refine the data structures and to add details with the definition of new events. The refinement of states is expressed in gluing invariant. Refinement of events consists in reinforcement of the guards and in the preservation of the gluing invariant. Event-B refinement relation is based on the alphabet translation which requires an explicit mapping between the components of the SDs (abstract and refined SD), which makes it possible to detect errors modeling and consequently to make the necessary corrections. Furthermore the relation refinement allows the verification of the termination of new events (their non-divergence).

To check the refinement relation between two given SDs (for instance $SD1$ and $SD2$), we first started by translating each SD into Event-B machine. In the refined machine, we found the same variables (representing the elements of the SD) as those of the abstract machine to which we can add new variables of the refined SD. In the invariant we express, in addition to the typing of each variable, the gluing invariant in which we

link the abstract variables to the concrete ones. The gluing invariant expresses the correspondence of the states related to the abstract and refined SD. We also defined the variant which guarantees the non divergence of new events introduced in the refined SD.

The proofs of the refinement relation and the verification of both obtained models allow to conclude or to check that $SD2$ is a correct refinement of $SD1$. Fig.20 represents the generic model for the checking of the correctness of the refinement between the two SDs.

ProB allows the checking of the refinement relation of Event-B specifications. It provides a multi-level animation facility to help to detect refinements' errors in a systematic way.

Acknowledgements We thank very much the anonymous reviewers for their time, their valuable comments and advice, which help us to considerably improve the article.

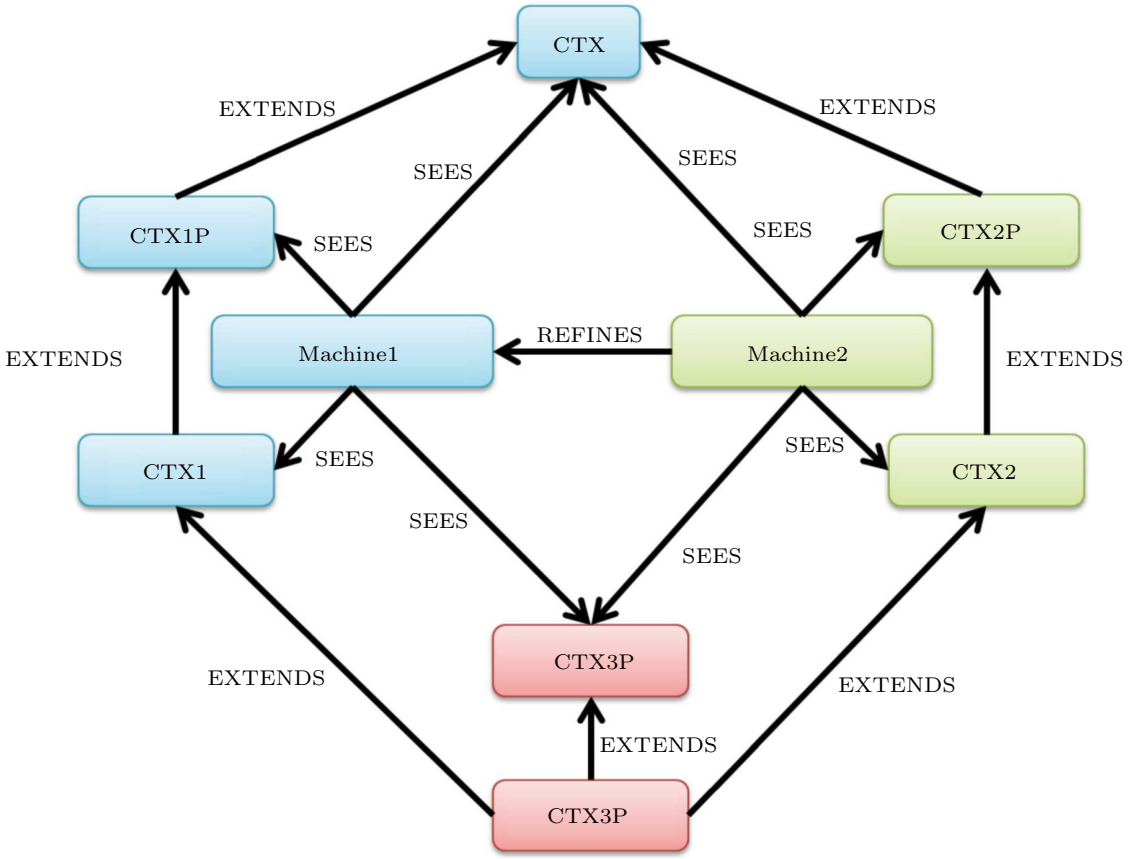

Fig.20. Generic Event-B architecture of refinement process.

References

[1] Micskei Z, Waeselynck H. The many meanings of UML 2 sequence diagrams: A survey. *Software & Systems Modeling*, 2011, 10(4): 489-514. DOI: 10.1007/s10270-010-0157-9.

[2] Pickin S, Jézéquel J M. Using UML sequence diagrams as the basis for a formal test description language. In *Proc. the 4th International Conference on Integrated Formal Methods*, April 2004, pp.481-500. DOI: 10.1007/978-3-540-24756-2_26.

[3] Störrle H. Trace semantics of interactions in UML 2.0. *J. Visual Languages and Computing*, 2004.

[4] Lund M S. Operational analysis of sequence diagram specifications [Ph.D. Thesis]. University of Oslo, 2008.

[5] André P, Rivière N, Waeselynck H. A Toolset for mobile systems testing. In *Proc. the 11th International Conference on Verification and Evaluation of Computer and Communication Systems*, August 2017, pp.124-138. DOI: 10.1007/978-3-319-66176-6_9.

[6] Mahe E, Gaston C, Gall P L. Revisiting semantics of interactions for trace validity analysis. In *Proc. the 23rd International Conference on Fundamental Approaches to Software Engineering*, April 2020, pp.482-501. DOI: 10.1007/978-3-030-45234-6_24.

[7] Dhaou F, Mouakher I, Attiogbé J C, Bsaies K. A causal semantics for UML 2.0 sequence diagrams with nested combined fragments. In *Proc. the 12th International Conference on Evaluation of Novel Approaches*

to Software Engineering, April 2017, pp.28-29. DOI: 10.5220/0006314100470056.

[8] Dhaou F, Mouakher I, Attiogbé J C, Bsaies K. An operational semantics of UML 2.X sequence diagrams for distributed systems. In *Proc. the 12th International Conference on Evaluation of Novel Approaches to Software Engineering*, April 2018, pp.158-182. DOI: 10.1007/978-3-319-94135-6_8.

[9] Dhaou F, Mouakher I, Attiogbé J C, Bsaies K. Guard evaluation and synchronization issues in causal semantics for UML 2.X sequence diagrams. In *Proc. the 13th Int. Conference on Evaluation of Novel Approaches to Software Engineering*, March 2018, pp.275-282. DOI: 10.5220/0006708102750282

[10] Abrial J R. Modeling in Event-B—System and Software Engineering. Cambridge University Press, 2010.

[11] Abrial J R, Butler M, Hallerstede S, Hoang T S, Mehta F, Voisin L. Rodin: An open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 2010, 12(6): 447-466. DOI: 10.1007/s10009-010-0145-y.

[12] Leuschel M, Butler M. ProB: An Automated analysis toolset for the B Method. *Int. J. Softw. Tools Technol. Transf.*, Springer-Verlag, 2008, 10(2): 185-203. DOI: 10.1007/s10009-007-0063-9.

[13] Boulanger J L. Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method. Wiley, 2014. DOI: 10.1002/9781119002727.

[14] Rasch H, Wehrheim H. Checking the validity of scenarios in UML models. In *Proc. the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, June 2005, pp.67-82. DOI: 10.1007/11494881_5.

[15] Knapp A, Wuttke J. Model checking of UML 2.0 interactions. In *Proc. the Workshops and Symposia at 2016 International Conference on Model Driven Engineering Languages and Systems*, October 2007, pp.42-51. DOI: 10.1007/978-3-540-69489-2_6.

[16] Lima V, Talhi C, Mouheb D, Debbabi M, Wang L, Pourzandi M. Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. *Electron. Notes Theor. Comput. Sci.*, 2009, 254: 143-160. DOI: 10.1016/j.entcs.2009.09.064.

[17] Cunha E, Custodio M, Rocha H, Barreto R. Formal verification of UML sequence diagrams in the embedded systems context. In *Proc. the 2011 Brazilian Symposium on Computing System Engineering*, November 2011, pp.39-45. DOI: 10.1109/SBESC.2011.18.

[18] Zhu M, Wang H, Liu X, Han X. Formal analysis of sequence diagram with time constraints by model transformation. *Int. J. Softw. Informatics*, 2012, 6(2): 327-357.

[19] Miyazaki H, Yokogawa T, Amasaki S, Asada K, Sato Y. Synthesis and refinement check of sequence diagrams. *IEICE Transactions on Information & Systems*, 2012, E95-D(9): 2193-2201. DOI: 10.1587/transinf.E95.D.2193.

[20] Remenska D, Willemse T A C, Templon J, Verstoep K, Bal H E. Property specification made easy: Harnessing the power of model checking in UML designs. In *Proc. the 34th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, June 2014, pp.17-32. DOI: 10.1007/978-3-662-43613-4_2.

[21] Muram F U, Tran H, Zdun U. Supporting automated containment checking of software behavioural models using model transformations and model checking. *Science of Computer Programming*, 2019, 174: 38-71. DOI: 10.1016/j.scico.2019.01.005.

[22] Lima L, Miyazawa A, Cavalcanti A, Cornélio M, Iyoda J, Sampaio A, Hains R, Larkham A, Lewis V. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, 2017, 16(3): 875-902. DOI: 10.1007/s10270-015-0492-y.

[23] Pan M, Chen S, Pei Y, Zhang T, Li X. Easy modelling and verification of unpredictable and preemptive interrupt-driven systems. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering*, May 2019, pp.212-222. DOI: 10.1109/ICSE.2019.00037.

[24] Chen X, Mallet F, Liu X. Formally verifying sequence diagrams for safety critical systems. In *Proc. the 14th Int. Symposium on Theoretical Aspects of Software Engineering*, December 2020, pp.217-224. DOI: 10.1109/TASE49443.2020.00037.

[25] Alur R, Holzmann G J, Peled D. An analyzer for message sequence charts. In *Proc. the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, March 1996, pp.35-48. DOI: 10.1007/3-540-61042-1_37.

[26] Tahir O, Sibertin-Blanc C, Cardoso J. A causality-based semantics for UML sequence diagrams. In *Proc. IASTED Int. Conference on Software Engineering*, February 2005, pp.106-111.

[27] Damchoom K, Butler M, Abrial J R. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Proc. the 10th International Conference on Formal Engineering Methods*, October 2008, pp.25-44. DOI: 10.1007/978-3-540-88194-0_5.

[28] Mouakher I. Case study for sequence diagram transformation in Event-B. Technical Report, 2021. https://www.dropbox.com/s/8oeuy4gwfsa8cqh/mainTR.pdf?dl=0.

Inès Mouakher received her Ph.D. degree in computer science from University of Nancy 2, France, in 2010. She joined University of Tunis El Manar, Tunis, as a lecture in 2004 and she became an assistant professor in computer science at University of Tunis El Manar in 2013. She is a member of the Laboratory of Computer Science Algorithmic and Heuristic Programming (LIPAH). Her research interests include formal methods, specification and verification, formal development (correction-by-construction), and distributed and concurrent system design. She published several peer-reviewed papers on these topics.

Fatma Dhaou is an assistant professor at University of Tunis El Manar, Tunis. She has been a member at the Laboratory of Computer Science Algorithmic and Heuristic Programming (LIPAH) of Faculty of Sciences of Tunisia and at the Laboratory of Digital Sciences (LS2N) of University of Nantes (France) since 2013. She received her Ph.D. degree in computer Science from University of Tunis El Manar, Tunis, in 2018. Her research interests include combining semi-formal language (UML) and formal method (Event-B method) for modelling, validation and verification of distributed system. She published several peer-reviewed papers on these topics.

J. Christian Attiogbé received his Ph.D. degree from University of Toulouse, France, in 1992, in computer science. He joined University of Nantes, Nantes, as an associate professor in 1994. His research interests include formal approaches for software modelling and analysis, correct-by-construction using refinement, and heterogeneous systems modelling. He published several peer-reviewed papers on these topics and also co-organised several workshops and conferences. He has been leading the Reliable Software Group at the Laboratory of Digital Sciences of Nantes (LS2N) since 2007, and had been the head of the Computer Science Department of the Nantes Institute of Technology at University of Nantes, Nantes (2010–2016). He is currently a professor of computer science at University of Nantes, Nantes.