View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

Flexible Modular Formalization of UML Sequence Diagrams

Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi
Politecnico di Milano – Dipartimento di Elettronica, Informazione e Bioingegneria
Via Golgi 42 – 20133 Milano, Italy
{luciano.baresi,mohammadmehdi.pourhashem,matteo.rossi}@polimi.it

ABSTRACT

UML Sequence Diagrams are one of the most commonly used type of UML diagrams in practice. Their semantics is often considered to be straightforward, but a more detailed analysis reveals diverse interpretations. These different choices must be properly supported by verification tools. This paper describes a formal framework for capturing semantic choices in a precise and modular way. The user is then able to select the semantics of interest, mix different interpretations, and analyze diagrams according to the chosen solution. This solution is supported by *Corretto*, our UML verification environment, to allow the user to play with different semantics and prove properties on Sequence Diagrams, accordingly.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Theory, Verification

Keywords

UML, metric temporal logic, formal semantics

1. INTRODUCTION

The detailed specification of a complex system would help the user understand its global behavior and become more confident about its correctness. Even if the particular domain would require precise and formal means, we often prefer to model a system through simple, visual notations, and UML is clearly one of the first choices in these years. UML provides both a significant set of (simple) visual notations and a great variety of modeling tools.

UML diagrams can be used to model (complex) systems and describe their peculiar aspects, but they would become even more interesting if one were able to use them to prove some properties formally. Rigorous analysis requires that the behavior of the modeling elements be stated unambiguously, but the complexity of the notation and its informal nature hamper its complete formalization. Several attempts have tried to formalize particular diagrams [5, 7], but these approaches are partial and only cover specific interpretations.

Supporting different (all) UML diagrams and diverse semantics is the main goal of our framework *Corretto*[1] [9]. In this paper we focus on Sequence Diagrams, which are often used to capture the most significant scenarios that describe how the components of a complex system interact.

The semantics of Sequence Diagrams is often assumed to be straightforward and shared, but the interpretation of the details makes the difference. UML 2 introduced combined fragments and more sophisticated control statements, and the interpretation of resulting diagrams has become even more complex. The different solutions that ascribe Sequence Diagrams with formal semantics tend to propose specific solutions [5, 7] with limited interoperability.

In contrast, this paper does not aim to introduce yet another semantics, but it studies the most significant proposals, organizes them into a single coherent framework, and proposes a solution to interpret Sequence Diagrams in a compositional and modular way. Users can decide the interpretations of the key aspects of their interest and the result is a complete and coherent semantics; then, provided some simple constraints are respected to avoid making inconsistent decisions, our framework accommodates all other aspects.

The proposed theoretical approach is implemented in *Corretto* and allows the user to easily play with the different semantics. The user can simulate the behavior of the diverse semantics and verify the satisfiability of the properties of interest. S/he can thus understand how the system behaves or work on the satisfiability of the properties and then obtain the guarantees the system must offer.

The rest of the paper is organized as follows. Section 2 informally introduces the possible interpretations of the different constructs of Sequence Diagrams. Section 3 presents our formalization framework and explains how to compose different, meaningful semantics. Section 4 sketches *Corretto* and explains how the user can work with the different semantics. Section 5 briefly surveys related approaches and Section 6 concludes the paper.

2. DIFFERENT INTERPRETATIONS

The interpretation of Sequence Diagrams is sometimes tricky. The OMG specification [1] does not define their

[1] https://github.com/mmpourhashem/CorrettoUML

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FormaliSE '14, June 3, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2853-1/14/06 ...$15.00.

semantics in detail to foster their adoption in different domains, but this allowed for diverse interpretations: for example, Micskei and Waeselynck [8] survey thirteen (sometimes similar) different alternatives.

We use the diagram of Figure 1 (`SDSearch`) to illustrate interpretations, explore ambiguities, and identify possible solutions. Informally, `app` is given a list of keywords, it pings `server1` and `server2` in parallel and passes the keywords to the first server that replies. `app` also periodically updates the `screen` with the results obtained from the server.

Most of the problems come with the interpretation of *combined fragments*. They allow the user to model complex interactions among objects in a succinct and organized manner. Each combined fragment depends on an *interaction operator*: `alt`, `opt`, `par`, `loop`, `break`, `seq`, `neg`, and others [1]. Every combined fragment comprises one or more *operands* and associated *guards* (nothing means *true*). Each operand, in turn, contains messages —and further combined fragments— that are processed according to the interaction operator associated with the enclosing combined fragment.

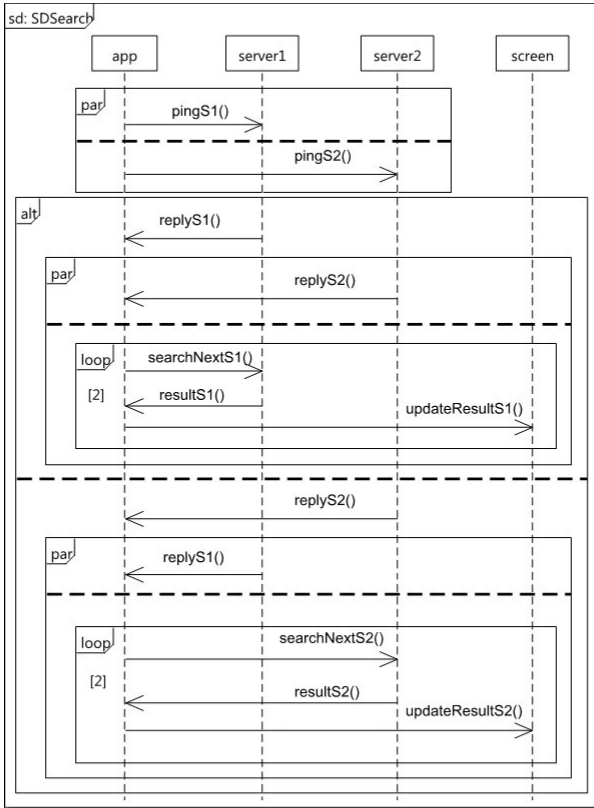

Figure 1: Example Sequence Diagram.

The first problem we address concerns the role played by the borders of a combined fragment. They can act as mere (graphical) containers of operands, whose execution order can then cross the borders, or be used to constrain the execution sequence of the messages of the different operands based on their position in the diagram. For example, in this latter case, it would not be possible for $replyS1$ to be sent before $pingS2$ is received. More rigorously[2], $T_1 = \{\langle !?pingS1, !pingS2\rangle, \langle !replyS1\rangle, \langle ?pingS2\rangle\}$ would be a trace fragment that satisfies the semantics of the diagram according to the former interpretation, but it would violate the latter since $!replyS1$ cannot happen before $?pingS2$.

The more general question is thus how the executions of the different combined fragments are blended (**[Combine]** in Figure 2). The OMG states that: *"The semantics of an interaction operand is given by its constituent interaction fragments combined by the implicit `seq` operation"*. This means that the combined semantics is defined through the semantics of *seq*, which obliges fragments to combine operands through *weak sequencing* (**WS** in Figure 2). If two events refer to the same lifeline, they follow the top-down order imposed by the lifeline. If they refer to different lifelines whose corresponding objects exchange messages (e.g., $pingS1$ and $replyS1$ between `app` and `server1`), then the send event on the first lifeline must also come before the receive event on the second lifeline. Hence, the order on a lifeline partially depends on the order of exchanged messages. Finally, if two events refer to independent lifelines, i.e., that do not exchange messages, they can appear in any order (e.g., $replyS1$ and $replyS2$ can happen in any order).

The OMG also states that borders do not impose any ordering constraint, and that the events before a combined fragment may happen after it and the events after the combined fragment may happen before it. [8] highlights that several proposed semantics, e.g., the work by Fernandes et al. [5], do not comply with what the OMG states. They impose an additional constraint on the weak sequencing that forces a *synchronous composition* (**SYNC**) among the different elements: the sequence of event occurrences respects the borders of combined fragments; i.e., events before a combined fragment happen before it and events after the combined fragment happen after its conclusion. This means that OMG would allow T_1 to be a fragment of a correct execution trace, while this further constraint would forbid it.

Combined fragments can also embed `loops`, but how the events in the different iterations are combined (**[Loop]**) is questionable. The occurrences in one iteration can: (a) be strictly separated from or (b) be interleaved with those of the others. Thus, the correctness of $T_2 = \{\langle !?searchNextS1_{It1}\rangle$, $\langle !?resultS1_{It1}\rangle$, $\langle !updateResultS1_{It1}\rangle$, $\langle !?searchNextS1_{It2}\rangle$, $\langle !?resultS1_{It2}\rangle$, $\langle ?updateResultS1_{It1}\rangle$, $\langle !updateResultS1_{It2}\rangle$, $\langle ?updateResultS1_{It2}\rangle\}$, where ItX means the xth iteration, is questionable.

The OMG states that the semantics of `loop` is equivalent to the recursive application of the semantics of `seq` (**WS**). These loops then become more permissive than those in programming languages —since their iterations cannot be intertwined. If one needed to mimic them, this would not be possible, in line with the OMG specification, even by using *general ordering*[3] in `loop`. Knapp and Wuttke [7] proposed to have weak sequencing for all fragments but loops. If iterations are to be combined synchronously (**SYNC**), T_2 would not be a correct execution trace, while it would be acceptable according to the pure OMG specification.

Even the evaluation of guards comes with some problems (**[When]**). One may say that (a) all lifelines evaluate guards

[2]The notation we use in this paper decomposes a message in the two events that correspond to sending (!) and receiving it (?). The shortcut *!?m* is equivalent to !m and ?m at the same time instant, and a pair of ⟨ ⟩ enclose the events that happen at the same time.

[3]A general ordering is a binary relation between events that describes that an event must occur before another in a trace.

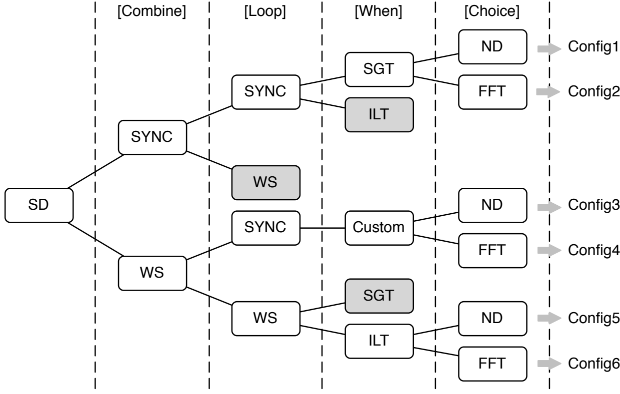

Figure 2: Different configurations obtained from the four choices.

at a single global time (**SGT**), or (b) each lifeline does it at particular independent local times (**ILT**). For example, T_1 corresponds to the case in which `app` and `server1` interact through the first operand of the `alt` fragment, and `server1` replies before `server2` has received the ping request. The correctness of this sequence depends on how the execution of composed fragments is interpreted: synchronous composition imposes that all guards be evaluated by each lifeline when the execution of the combined fragment starts. Weak sequencing would allow the first lifeline entering the combined fragment to evaluate guards and decide which operand to execute, if needed. All other lifelines must then obey the decision taken by the first lifeline, even if meanwhile guards have changed.

Besides the evaluation time, more than one guard may be true in an `alt` fragment ([**Choice**]). If this is the case, one could: (a) pick one nondeterministically (**ND**), or (b) select the first true guarded operand from the top (**FFT**). The OMG only states that: *"At most one of the operands will be chosen"*, but there is no advice on the selection in case of alternatives. Most of the known interpretations simply pick one true-guarded operand nondeterministically. Others simply select the first one from the top. The latter choice is the way to go if one adopted the implicit assumption that operands are ordered top-down according to their priority.

All aforementioned problems, and choices, must be properly addressed, and one single solution be identified. There might be different ways of presenting a scenario by means of Sequence Diagrams, but there must always be one and only one interpretation of the diagram.

To this end, the decision tree of Figure 2 summarizes our analysis and identifies six possible coherent and complete interpretations given the different choices and their mutual constraints. Gray nodes identify inconsistent interpretations, and therefore the derivation is aborted. For example, a coherent interpretation cannot use **SGT** and **WS**: a single global time cannot be selected when fragments are composed according to weak sequencing. Similarly, **WS** for loops is not compatible with **SYNC**, as composing fragments through synchronous composition and iterations in loops through weak sequencing would be incoherent.

Of the six resulting configurations, **Config5** is the only one that fully complies with OMG's specification. **Config3** and **Config4** are special cases: they use a single global time to evaluate the guards associated with loops, and independent local times on each lifeline for the other fragments.

3. FORMAL SEMANTICS

This section presents our modular, unifying approach for formalizing the different semantic variants described in Section 2. The approach is based on temporal logic and builds on previous work [4] on the formalization and verification of UML diagrams —not limited to Sequence Diagrams. Temporal logic is well suited for our purposes, because it allows us to focus on the different aspects of the semantics by using separate groups of formulae, and then obtain the complete formalization through simple logic conjunction.

At the basis, given a Sequence Diagram $\mathcal{S}$, we introduce a set of logic propositions that capture the behavior of the elements of $\mathcal{S}$. For example, proposition $\mathcal{S}_{Start}$ describes the start of the execution of diagram $\mathcal{S}$. If $\mathcal{S}$ includes a message m that is exchanged between two objects, propositions m_{Start} and m_{End} correspond to the message being respectively sent and received, and so on. Every element of $\mathcal{S}$ that is relevant for its semantics has a corresponding proposition, including the borders of combined fragments (to represent when they are crossed during the execution of $\mathcal{S}$), their guards, etc. Then, for each aspect of the semantics of a Sequence Diagram $\mathcal{S}$, we introduce logic formulae that capture it. For example, given a lifeline $\mathcal{L}$ of $\mathcal{S}$, we introduce a set of formulae, say $Ord_{\mathcal{L}}$, that capture the ordering of events along the lifeline. If $\mathcal{L}_1 \dots \mathcal{L}_n$ are all the lifelines of $\mathcal{S}$, the complete formalization of the ordering of events through $\mathcal{S}$ is given by $Ord_{\mathcal{S}} = \bigwedge_{i=1}^{n} Ord_{\mathcal{L}_i}$.

Now, consider a combined fragment $\mathcal{C}$ that spans different lifelines, such as those depicted in Figure 1. A set of formulae, say $Sem_{\mathcal{C}}$, captures the semantics of $\mathcal{C}$, including the conditions under which $\mathcal{C}$ is entered (for example, whether all lifelines enter the fragment at the same time or not) or exited. Formula $Ord_{\mathcal{S}} \wedge Sem_{\mathcal{C}}$ constrains the ordering of events to obey the semantics of $\mathcal{C}$. Note that formulae $Ord_{\mathcal{S}}$ and $Sem_{\mathcal{C}}$ will include common propositions such as, for example, the proposition corresponding to $\mathcal{C}$ being entered, since it is also an event that occurs along the lifelines. These *shared propositions* provide a form of "synchronization" among the various parts of the formalization, as their behavior is constrained by several concurring aspects.

The different variants presented in Section 2 give rise to different ways of formalizing the various aspects of a Sequence Diagram $\mathcal{S}$. For example, the semantics of combined fragment $\mathcal{C}$ is different depending on whether a **SYNC** or a **WS** semantics is used. If we indicate by $Sem_{\mathcal{C}}^{\mathrm{SYNC}}$ the semantics of $\mathcal{C}$ in the former case and by $Sem_{\mathcal{C}}^{\mathrm{WS}}$ the one in the latter, then changing from one to the other is as simple as substituting $Sem_{\mathcal{C}}^{\mathrm{WS}}$ for $Sem_{\mathcal{C}}^{\mathrm{SYNC}}$ in $Ord_{\mathcal{S}} \wedge Sem_{\mathcal{C}}^{\mathrm{SYNC}}$.

In the rest of this section we provide some details of the formalization of the semantic variants presented in Section 2. The approach exploits the metric temporal logic TRIO [6], which allows users to express timing properties of systems, including real-time ones. In fact, though in this paper we focus on the flexibility and modularity aspects of our approach rather than the real-time ones, we allow users to express and verify metric properties such as "message m_1 will be followed, within 3 time units, by message m_2". TRIO adopts a linear notion of time, and can be used to express properties over both discrete and continuous temporal domains; here we only focus on the former.

Table 1: TRIO operators used in this paper.

Operator	Definition	Meaning
$\text{Futr}(\phi,d)$	$d>0 \wedge \text{Dist}(\phi,d)$	ϕ occurs d instants in the future
$\text{Past}(\phi,d)$	$d>0 \wedge \text{Dist}(\phi,-d)$	ϕ occurred d instants in the past
$\text{Alw}(\phi)$	$\forall t(\text{Dist}(\phi,t))$	ϕ always holds
$\text{Lasts}(\phi,d)$	$\forall(0<t<d \Rightarrow \text{Futr}(\phi,t))$	ϕ holds for the next d time units
$\text{Lasted}(\phi,d)$	$\forall(0<t<d \Rightarrow \text{Past}(\phi,t))$	ϕ held for the last d time units
$\text{Until}(\phi,\psi)$	$\exists t(\text{Futr}(\psi,t) \wedge \text{Lasts}(\phi,t))$	ψ will occur and ϕ will hold until then
$\text{Since}(\phi,\psi)$	$\exists t(\text{Past}(\psi,t) \wedge \text{Lasted}(\phi,t))$	ψ occurred and ϕ held since then

Table 1 defines some typical TRIO operators in terms of the basic Dist operator, where $\text{Dist}(\phi,d)$ means that ϕ holds at the instant exactly d time units from the current one. There are also variants of the operators that include one or both of the endpoints of an interval; for example, $\text{Lasts}_{\text{ei}}(\phi,d) = \forall(0<t\leq d \Rightarrow \text{Futr}(\phi,t))$.

3.1 Basic Modules

We start by presenting the formulae that capture cross-cutting aspects of the semantics of Sequence Diagrams.

3.1.1 Order

The formalization of the ordering between two events is the simplest part of the semantics, and it is the building block for formalizing partial and general ordering in Sequence Diagrams. However, even the simple notion of "ev_1 is followed by ev_2" raises some questions about the validity of traces. What if we had ev_1 and no ev_2 afterwards? Is ev_1 alone valid? Dually, what if we had ev_2 without a previous ev_1? Can they occur at the same time? Several interpretations are possible.

Mono-Directional Order.

Event ev_2 may be triggered by the occurrence of event ev_1 under certain circumstances. In this case we formalize that, after ev_1, ev_2 occurs, provided that a suitable guard holds when ev_1 occurs. However, if the ordering is only *mono-directional*, an occurrence of ev_2 does not imply a previous ev_1. Formula (1) captures this notion of ordering (where *guard* and *exception* are placeholders for sub-formulae that are introduced below).

$$ev_1 \wedge \text{guard} \Rightarrow \begin{pmatrix} \text{Until}_{\text{ei}}((\neg ev_1 \wedge \neg ev_2), \text{exception}) \vee \\ \text{Until}_{\text{ei}}((\neg ev_1 \wedge \neg \text{exception}), ev_2) \end{pmatrix} \quad (1)$$

The next formula, instead, defines that event ev_2 cannot occur at the same time as its trigger (ev_1):

$$ev_1 \wedge \text{guard} \Rightarrow \neg ev_2. \quad (2)$$

Following [4], we write the modular semantics of Sequence Diagrams to fit it into a larger UML semantics, which also includes other diagrams such as Interaction Overview Diagrams. Hence, we allow a Sequence Diagram to be stopped at any time during its execution, provided a suitable event (an *exception*, for example a timeout) occurs. This is captured by Formula (1), which states that ev_2 occurs after $ev_1 \wedge \text{guard}$ holds, unless an exception intervenes. Finally, we introduce the following abbreviation:

$$\textbf{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) \stackrel{\text{def}}{=} \begin{cases} (1) & \text{if isConcurrent} = true \\ (1) \wedge (2) & \text{otherwise} \end{cases}$$

Mono-Directional Reverse Order.

When event ev_2 is necessarily triggered by event ev_1, but not all occurrences of ev_1 produce ev_2, then we formalize that the former has to be preceded by the latter, but not the converse. This is captured by the following formula:

$$ev_2 \Rightarrow \text{Since}_{\text{ei}}((\neg ev_2 \wedge \neg \text{exception}), (ev_1 \wedge \text{guard}))\,. \quad (3)$$

Abbreviation **OrderMonoDRev** is similar to **OrderMonoD** and is omitted for brevity.

Bidirectional Order.

This is the conjunction of the two previous cases: ev_1 occurs if, and only if, ev_2 also occurs later (unless an exception occurs before):

$$\begin{aligned} &\textbf{Order}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur}) \stackrel{\text{def}}{=} \\ &\quad \textbf{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur}) \wedge \\ &\quad \textbf{OrderMonoDRev}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur}) \end{aligned}$$

3.1.2 Borders

Each Sequence Diagram, combined fragment, lifeline, operand of combined fragment and message has a beginning, an end, and a duration throughout which the element is "active" (e.g., a message that was sent and has yet to be received, or an operand that was entered and not exited); hence, we treat them as "modules", each with its own semantics. For each module M we introduce three predicates M, M_{Start} and M_{End} capturing, respectively, the module being "active", its start, and its end. Formulae (4)-(5) formalize their behavior.

$$\text{M} \Leftrightarrow \text{Since}_{\text{ei}}(\neg \text{M}_{End} \wedge \neg \text{exception}, \text{M}_{Start}) \quad (4)$$

$$\text{M}_{Start} \Rightarrow \text{Until}_{\text{ei}}(\neg \text{M}_{Start}, \text{M}_{End} \vee \text{exception}) \quad (5)$$

Formula (4) defines that a module is active from its start until its end (which could occur with an exception in the enclosing module), and Formula (5) states that after a module starts, it must end, possibly because of an exception, and there is no further start until then. Abbreviation **Borders** captures the semantics of the start and end of a module:

$$\textbf{Borders}(\text{Module}, \text{exception}) \stackrel{\text{def}}{=} (4) \wedge (5).$$

3.1.3 Auxiliary Operators

It is useful to introduce, as abbreviations, variations of the "eventually" TRIO operators (SomF and its past counterpart SomP) that have the meaning of "eventually during the current execution of the enclosing fragment" (which could be a combined fragment or possibly the whole Sequence Diagram). We call these abbreviations SomFIn_{i} and SomPIn_{i}, where the subscript i means that the current instant is included. They are defined as follows (we only show the definition of SomFIn_{i}, the other being dual):

$$\text{SomFIn}_{\text{i}}(ev_1, \text{enclosingModule}) \stackrel{\text{def}}{=} \neg \text{Until}_{\text{ii}}(\neg ev_1, \text{enclosingModule}_{End})\,.$$

3.2 Combined Fragments

As Figure 2 shows, to define the semantics of Sequence Diagrams there are four choices to make. In the rest of this section we describe how these choices impact different pieces of the formal semantics, and we show how they are combined to form a complete semantics of diagrams. Due to lack of space, we mostly focus on the `alt` combined fragment, and we provide highlights of `par` and `loop`.

3.2.1 Alternative

Let us consider an `alt` fragment *Alt* that is part of a Sequence Diagram S. We first consider the semantics when the choices for [**Combine**] and [**Choice**] are, respectively, **WS** and **ND**. Then, we show how the semantics changes for different choices. The following formula —introduced in Section 3.1.2 —defines the basic behavior of the start and end propositions of *Alt*, and it identifies S_{Stop} as the event that can interrupt the execution of the combined fragment.

$$\mathbf{Borders}(Alt, S_{Stop}) \tag{6}$$

Formula (7) defines that fragment *Alt* is activated as soon as one of the n lifelines it spans (which are indicated in the formula by Alt_L^i, with $1 \leq i \leq n$) enters the fragment (which is represented by proposition $Alt_L^i_{Start}$).

$$(\bigvee_{i=1}^{n} Alt_L^i_{Start} \Rightarrow Alt) \wedge (Alt_{Start} \Rightarrow \bigvee_{i=1}^{n} Alt_L^i_{Start}) \tag{7}$$

To capture the different semantics, we introduce propositions to represent when a lifeline L_j enters the combined fragment *Alt* ($Alt_L^j_{Start}$) and when it enters one of its m operands OP^i ($Alt_OP^i_L^j_{Start}$). The two events are of course related, as entering operand OP^i occurs if, and only if, the combined fragment was previously entered. This is captured by Formula (8), in which the guard Alt_OP^i implies that the lifeline can enter only the active operand; that is, lifelines that enter the combined fragment at different instants must enter the same operand.

$$\bigwedge_{i=1}^{m}\bigwedge_{j=1}^{n} \mathbf{Order}(Alt_L^j_{Start}, Alt_OP^i_L^j_{Start}, Alt_OP^i, S_{Stop}, true) \tag{8}$$

Formula (9), instead, defines that the exit operand OP^i of a lifeline L^j of fragment *Alt* is followed by L^j exiting *Alt* itself.

$$\bigwedge_{i=1}^{m}\bigwedge_{j=1}^{n} \mathbf{OrderMonoD}(Alt_OP^i_L^j_{End}, Alt_L^j_{End}, true, S_{Stop}, true) \tag{9}$$

Formula (10) captures the semantics of the *else* operand of *Alt*, which is entered if, and only if, the guards of all other operands are false when the combined fragment is entered.

$$Alt_OP^{Else}_{Start} \Leftrightarrow (\neg(\bigvee_{i=1}^{m} Alt_Guard^i) \wedge Alt_{Start}) \tag{10}$$

Formula (11) defines that when combined fragment *Alt* starts being executed, exactly one of its operands (possibly the *else* operand) starts its execution.

$$Alt_{Start} \Rightarrow \bigvee_{i\in[1,m]\cup Else} (Alt_OP^i_{Start} \wedge \bigwedge_{\substack{j\in[1,m]\cup Else \\ j\neq i}} \neg Alt_OP^j_{Start}) \tag{11}$$

Formula (12) formalizes that the combined fragment ends when one of its operands (which, by the constraints above, must be the one that was chosen for execution) terminates.

$$\bigvee_{i\in[1,m]\cup Else} Alt_OP^i_{End} \Leftrightarrow Alt_{End} \tag{12}$$

Finally, Formula (13) states that an operand is chosen only if its guard is true when *Alt* starts its execution.

$$\bigwedge_{i=1}^{m}(Alt_OP^i_{Start} \Rightarrow Alt_{Start} \wedge Alt_Guard^i) \tag{13}$$

The semantics of the *Alt* combined fragment is completed by adding the formulae defining the behavior of each operand, but we skip this for brevity.

Finally, we define the following abbreviation (where F_i is the ith formula in the paper):

$$\mathbf{AltCF}(\text{Alt}, \mathbf{WS}, \mathbf{ND}) \stackrel{\text{def}}{=} \bigwedge_{i\in[6,13]} \text{Alw}(F_i).$$

If the user selects **FFT** as semantic choice instead of **ND**, Formula (11) is replaced by the following one, which selects an operand OP^i only if its guard holds, and $Guard^j$ is false for all $j < i$:

$$\bigwedge_{i=1}^{m} Alt_OP^i_{Start} \Leftrightarrow Alt_{Start} \wedge Alt_Guard^i \wedge \neg \bigvee_{j=1}^{i-1} Alt_Guard^j \tag{14}$$

In this case we have the following abbreviation:

$$\mathbf{AltCF}(\text{Alt}, \mathbf{WS}, \mathbf{FFT}) \stackrel{\text{def}}{=} \bigwedge_{i\in[6,10]\cup[12,14]} \text{Alw}(F_i).$$

If the choice of [**Combine**] is **SYNC** instead of **WS**, the admissible traces are a subset of those allowed by **WS**, since there are additional constraints on the start and end of the combined fragment. More precisely, in this case the following formula is added:

$$(Alt_{Start} \Leftrightarrow \bigwedge_{i=1}^{n} Alt_L^i_{Start}) \wedge (Alt_{End} \Leftrightarrow \bigwedge_{i=1}^{n} Alt_L^i_{End}) \tag{15}$$

and the corresponding abbreviation becomes the following:

$$\mathbf{AltCF}(\text{Alt}, \mathbf{SYNC}, \mathbf{ND}) \stackrel{\text{def}}{=} \bigwedge_{i\in[6,13]\cup\{15\}} \text{Alw}(F_i).$$

Another possibility to define $\mathbf{AltCF}(\text{Alt}, \mathbf{SYNC}, \mathbf{ND})$ is to replace each proposition $Alt_L^i_{Start}$ (resp. $Alt_L^i_{End}$) with Alt_{Start} (resp. Alt_{End}), and to eliminate Formula (7), which is in this case subsumed by the others.

3.2.2 Parallel

For a `par` combined fragment the only meaningful semantic choice is between **WS** and **SYNC**. As mentioned above, the semantics of **SYNC** can be obtained by adding constraints to the **WS** semantics, so we focus on the latter.

When a `par` combined fragment P is activated (i.e., when one of the lifelines enters the fragment first) the guards of its operands are evaluated and for those that are true the corresponding operands are activated. When another lifeline enters the fragment, it is allowed to start executing its events in the activated operands in parallel. If all guards evaluate to false, the `par` fragment collapses over all lifelines, that is, its start coincides with its end, as defined by Formula (16).

$$P_{Start} \wedge \neg \bigvee_{i=1}^{m} P_Guard^i \Rightarrow \bigwedge_{i=1}^{n} \text{SomFIn}_{\text{i}}(P_L^i_{Start} \wedge P_L^i_{End}, P)\,. \tag{16}$$

Conversely, Formula (17) states that the end point of a lifeline in a `par` fragment should occur either at the same time of its start point (in case all guards are false), or when it leaves the last operand of the fragment.

$$\bigwedge_{i=1}^{n} (P_L^i_{End} \Rightarrow$$
$$(P_L^i_{Start} \wedge \text{SomPIn}_\text{i}(P_{Start} \wedge \neg \bigvee_{j\in[1,m]} P_Guard^j, P)) \vee$$
$$(\bigvee_{j\in[1,m]} P_OP^j_L^i_{End} \wedge \bigwedge_{j\in[1,m]} (P_OP^j_L^i \Rightarrow P_OP^j_L^i_{End})) \quad (17)$$

The full semantics of fragment P is then obtained by the conjunction of Formulae (16) and (17), plus others similar to those presented in Section 3.2.1.

3.2.3 Loop

A `loop` fragment with a *min* and a *max* number of iterations has the same behavior as a *min* number of `seq` fragments (i.e., it would behave according to the sequential semantics of Sequence Diagrams), followed by a $max - min$ number of `opt` fragments (which are a special case of the `alt` fragment). Each of the above fragments contains the same operand, indicated in the following as *Loop_OP*, which is repeatedly activated during the loop. The options for the semantics of loops (see Figure 2) are **WS** or **SYNC**.

We build the **WS** semantics by reducing a loop to a sequence of `seq` and `opt` fragments, as explained above. Since in a **WS** semantics different lifelines can be in different iterations of the loop at the same time (e.g., lifeline L^1 is in iteration 1, but L^3 is in iteration 2), we introduce different sets of predicates for each iteration. Then, the semantics of each iteration is produced similarly to the case presented in Section 3.2.1, using its specific propositions. In addition, the end events of each iteration are linked to the start events of the next one by the ordering mechanisms of Section 3.1.1.

When the chosen semantics for loops is **SYNC**, the formalization can be simplified. Since all lifelines execute the same iteration at the same time, we introduce only one set of propositions for operand *Loop_OP*. We keep track the current iteration through propositions that describe a finite counter C, whose values range from 0 to *max*.

For example Formula (18) defines that if, at the end of an iteration, the guard of the loop is false and the number of iterations is over or at the minimum, then the execution of the loop ends.

$$Loop_OP_{End} \wedge C \geq min \wedge \neg Loop_Guard \Rightarrow Loop_{End} \quad (18)$$

Table 2: Experimental results with *Corretto* (times are in seconds).

		Config1	Config2	Config3	Config4	Config5	Config6
T1	Result	UNSAT	UNSAT	SAT	SAT	SAT	SAT
	Time	6	6	31	27	40	33
T2	Result	UNSAT	UNSAT	UNSAT	UNSAT	SAT	SAT
	Time	6	6	15	14	40	38
P1	Result	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
	Time	14	11	41	36	64	46
P2	Result	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
	Time	13	11	57	37	53	41
TC	Result	UNSAT	UNSAT	UNSAT	UNSAT	SAT	SAT
	Time	12	9	30	24	42	34

Formula (19) states that, if the guard holds at the end of the iteration, we have a nondetrministic choice of continuing with the loop or ending it ($\oplus$ is the exclusive or).

$$Loop_OP_{End} \wedge C \geq min \wedge C < max \wedge Loop_Guard \Rightarrow \text{Futr}(Loop_OP_{Start}, 1) \oplus Loop_{End} \quad (19)$$

Finally, Formula (20) states that when the maximum number of iterations is reached, the execution ends.

$$Loop_OP_{End} \wedge C = max \Rightarrow Loop_{End} \quad (20)$$

The complete semantics of the `loop` fragment is produced from the basic building blocks in a similar way as before.

4. INITIAL EVALUATION

The proposed modular semantics for Sequence Diagrams has been implemented within *Corretto*, our toolset for the formal verification of UML models. The tool is built on top of the eclipse-based Papyrus UML modeler[4] and translates UML diagrams into TRIO formulae, which are then fed to $\mathbb{Z}$ot [2] for their formal verification.

We used the resulting toolset for verifying the Sequence Diagram of Figure 1. This is only a first evaluation and a more complete assessment is in progress. We compared the different configurations, and present some of their discrepancies, in three ways:

(i) We injected the previously introduced traces T_1 and T_2 as additional specifications for the system and checked whether the system were satisfiable.

(ii) We asserted properties P_1 and P_2, introduced below, to check whether they held for the system.

(iii) We imposed the time constraints introduced below on the system to study some real-time requirements.

Table 2 summarizes the results produced by the tool and the average times needed to obtain them (although the difference between min and max values is always less than a second).

$\mathbb{Z}$ot allows one to inject a partial (or complete) trace T and check whether it complies with the model. If it does, $\mathbb{Z}$ot returns a complete version of T (SAT) as output, otherwise the output is empty (UNSAT). We used this feature to analyze traces T_1 and T_2 and confirm the informal results already introduced in Section 2: T_1 is not satisfiable for those configurations that adopt the synchronous composition of combined fragments, whereas T_2 is satisfiable only for **Config5** and **Config6**, that is, for those configurations that exploit weak sequencing for composing fragments.

Property P_1 is defined as follows: *"If `server1` sends a reply to `app` one time instant before `server2` does, `app` will not send any search request to `server2` during the current execution of diagram `SDSearch`".* Its formal equivalent is

$$P_1 \overset{\text{def}}{=} \text{Alw}\begin{pmatrix} \text{replyS1}_{Start} \wedge \text{Futr}(\text{replyS2}_{Start}, 1) \Rightarrow \\ \neg\text{SomFIn}_\text{i}(\text{searchNextS2}_{Start}, \text{SDSearch}) \end{pmatrix}.$$

Since in this specific Sequence Diagram no guard is defined for the two operands of `alt`, both guards are implicitly true, and according to the **FFT** variant of **[Choice]**, the first operand is always chosen. Consequently, for configurations in line with **FFT**, the second operand of `alt` never gets

[4] `http://www.eclipse.org/papyrus/`

activated, and the property holds for them. Among the other configurations, it holds for **Config1** (since the two replies are separated by a border, they can only occur in the order depicted in the diagram), but not for **Config3** and **Config5**. In the counterexamples provided by *Corretto* for the two cases in which the property fails, $T_3 = \{\langle !replyS1\rangle, \langle !replyS2\rangle, \langle ?replyS2\rangle, \langle ?replyS1\rangle, \langle !replyS1\rangle, \langle !searchNextS2\rangle\}$ is part of the traces, because the sending of $replyS1$ and $replyS2$ can be in any order thanks to the choice of weak sequencing.

P_2 is *"If* `app` *receives a reply from* `server1` *one time instant before receiving one from* `server2`, `app` *will not send any search request to* `server2` *during* `SDSearch`*"*; its formal equivalent is very similar to the one of P_1, and is omitted for brevity. The property holds for every configuration; that is, for all traces, no matter the interpretation, if there is *?replyS1* at time instant t and *?replyS2* at $t+1$, then there is no *!searchNextS2* at any future time instant in the trace.

Corretto also supports time constraints. Here the focus is not on their formalization; rather, we exploit this capability to study the real-time-related behavior of `SDSearch` and the impact of the different configurations. We set the number of loop iterations to 2 in order to ease the understanding of the diagram. The three time constraints state that the transmission of messages `updateResultS1` and `updateResultS2`, that is, the time difference between sending and receiving them, takes at least one time instant and that the whole `SDSearch` takes no longer than 8 time instants. This is captured by the following annotations added to the Sequence Diagram:

$$(@updateResultS1_{End} - @updateResultS1_{Start}) \geq 1$$
$$(@updateResultS2_{End} - @updateResultS2_{Start}) \geq 1$$
$$(@SDSearch_{End} - @SDsearch_{Start}) \leq 8$$

Table 2 shows that these constraints can only be satisfied if **Config5** and **Config6** are adopted. These are the cases where weak sequencing is used to manage loop iterations.

Even if preliminary, our results witness the importance of the semantics adopted for reasoning on Sequence Diagrams. It cannot remain implicit, and the designer must be both fully aware of the subtle choices and be able to set them.

5. RELATED WORK

Sequence Diagrams have been formalized in different ways because of the complexity of UML, their possible different uses, and the incompleteness of the OMG specification. Micskei and Waeselynck [8] investigated almost all proposed interpretations and categorized them based on their semantic choices. Only few of them have been introduced by their authors in a way that is rigorous and amenable for formal verification. Due to this limitation, and because of lack of space, this section focuses on the proposals that address the formalization of composed fragments, and it explains how these interpretations can be obtained through our modular solution.

Fernandes et al. [5] exploit colored Petri nets to explain their semantics, where combined fragments execute in a synchronized way and loop iterations are strictly separated. Since the `alt` interaction operator is more permissive than *if-then-else* in programming languages[5], they constrain it and assume that the designer always assign disjoint guards to its operands. Given these assumptions, and the rest of the choices they make, their semantics would fit either **Config1** or **Config2** of Figure 2.

Knapp and Wuttke [7] propose a translation of Sequence Diagrams into automata. Model checking can then be used to verify the properties of interest against the synthesized operational description. Besides proposing a formal semantics for the main interaction operators, they also introduce a new one called `sloop`, that is, a loop whose iterations are strictly separated from one another. Putting `sloop` aside, their semantics complies with OMG specification, which is equivalent to our **Config5**. However, if the designer used `sloop` instead of `loop`, then this semantics would correspond to our **Config3**.

Our configurable semantics answers some of the recommendations proposed by Atlee et al. [3] for improving the usability of formal methods. They recommend that transformation tools be semantically configurable, and that the commonalities among different implementations be factored out and exploited. This is exactly the key goal of this work, and more in general of *Corretto*, our *customizable* verification environment for UML.

6. CONCLUSIONS AND FUTURE WORK

The paper presents a comprehensive, modular semantics for UML Sequence Diagrams that accommodates existing interpretations. Besides the theoretical setting, the paper also proposes a first evaluation of the six possible configurations we have identified through *Corretto*, our UML verification toolset. The plan for the future is to cover the whole set of combined fragments and keep studying the different interpretations ascribed to the diverse UML diagrams.

[5]More than one guard can be true at the same time.

7. REFERENCES

[1] Object Management Group: UML 2.4. 1 Superstructure Specification, formal/2011-08-06.

[2] The Zot Satisfiability Checker. `zot.googlecode.com`.

[3] J. Atlee, S. Beidu, N. Day, F. Faghih, and P. Shaker. Recommendations for Improving the Usability of Formal Methods for Product Lines. In *Formal Methods in Software Engineering*, pages 43–49, 2013.

[4] L. Baresi, A. Morzenti, A. Motta, and M. Rossi. A Logic-based Semantics for the Verification of Multi-diagram UML Models. *ACM Soft. Eng. Notes*, 37(4):1–8, 2012.

[5] J. Fernandes, S. Tjell, J. Baek Jorgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *Proc. of SCESM*, May 2007.

[6] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi. *Modeling Time in Computing*. Springer, 2012.

[7] A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. In *Models in Software Engineering*, pages 42–51. 2007.

[8] Z. Micskei and H. Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[9] A. Motta. Logic-based verification of multi-diagram UML models for timed systems. Ph.D. thesis, Politecnico di Milano, 2013.