Formalize UML 2 Sequence Diagrams

Department of Computer Science
University of Texas at San Antonio
Technical Report: CS-TR-2008-013
Hui Shen, Aliya Virani, and Jianwei Niu
{hshen, avirani, niu}@cs.utsa.edu

Abstract— **UML 1 sequence diagrams have been widely employed for modeling software requirements and design. UML 2 introduced many new features, such as Combined Fragments, to make sequence diagrams more expressive than UML 1. However, the lack of formal semantics descriptions of these features makes it difficult for practitioners and tool builders to construct and analyze sequence diagrams that specify high assurance systems. In previous work, we presented a formalism, template semantics, for describing the operational semantics of behavioral notations. In this paper, we adapt template semantics to describe the combined fragments and other constructs of sequence diagrams. We believe that formalizing the semantics of sequence diagrams is an important step towards synthesizing multiple sequence diagrams into behavioral models (i.e., state machines) and formally analyzing them.**

Key words: formal methods, semantics, sequence diagrams, combined fragments

I. Introduction

The Unified Modeling Language (UML) is a collection of modeling notations for specifying different artifacts, such as requirements and design, of software systems. Sequence diagram is one of the key notations of UML and serves as a well-accepted media among software developers, stakeholders, and tool builders. The appeal can be attributed to the intuitive nature of its graphical representation and its capability to capture a scenario of how some of the functionality might be used, or how users interact with other components. In addition, models enable us to detect errors during the early stages of software development so as to improve the system quality. Yet, for the investment of constructing sequence diagrams, analysis of them remain largely limited to human inspection, which can be error-prone. While appropriate for some projects, high assurance systems require substantial proof to demonstrate that they perform reliably and safely. Formal analysis techniques may provide evidence that desired properties hold, and they often require executable models in notations with formal semantics. Unfortunately, only informal description of sequence diagrams is provided by OMG in UML superstructure [17], which is in a scattered manner across hundreds of pages of technical specifications and terms.

This paper formalizes the semantics of UML 2 sequence diagrams in a larger effort to provide practitioners with the ability to build analysis tools. A sequence diagram describes an interaction between actors and other entities of a system to obtain desired behavior or services by transmitting a sequence of messages. UML 2 adds some new structured control constructs to a sequence diagram, such as InteractionUse and CombinedFragment, to express concurrent message exchanges. However, the lack of precise descriptions makes it difficult for practitioners to understand and use these new features, or to disclose faults in the models. To alleviate these problem, we construct formal semantics for sequence diagrams using template semantics [15], [14].

In previous work, we develop template semantics, which is a formalism to structure the operational semantics of a family of specification notations, such as UML stateMachines, statecharts [7], [8], process algebras[12], [10], and Petri Nets[19]. In template semantics, the basic computation model is hierarchical transition systems (HTSs), which are extended state machines. Communication, concurrency, and synchronization among HTSs are achieved via composition operators. The template semantics of HTS is defined as a sequence of steps and the composition operators are expressed as logic constraints on how HTSs execute concurrently.

We adapt concepts from template semantics to formalize UML 2 sequence diagrams. We deconstruct the syntactic elements of a sequence diagram and map them to HTSs, whose semantics are represented as step relations. In the context of sequence diagrams, a step is generally captured by sending or receiving of messages. We extend template semantics composition operators to define the compositional semantics of InteractionUse and CombinedFragment. These semantics definitions will make it easier for practitioners to understand the sequential and concurrent natures of interactions, so as to construct sequence diagrams. We also believe that formalizing the semantics of a sequence diagram will enable not only the synthesis of more succinct behavior models from multiple sequence diagrams, but also a means to build analysis tools to detect subtle errors in models.

The rest of the paper is structured as follows. Section 2 briefly discusses the UML 2 sequence diagram's syntax and semantics provided by OMG. Section 3 provides background information on template semantics. In section 4, we describe how to syntactically map sequence diagram to HTSs. Section 5 presents the formal semantics of a sequence diagram using template semantics. Section 6 provides related works. Finally, we conclude with Section 7 and suggest future work.

II. UML 2 Sequence Diagrams

Sequence diagram expresses interactions between objects by exchanging Messages. In this section we briefly describe its syntax and semantics provided by OMG.

A. Metamodel

We provide UML 2 sequence diagram a metamodel, which graphically displays the abstract syntax in terms of class diagram to ease users' effort required to understand the notation. The metamodel complies with the interaction metamodel provided by OMG (pages 458-465 in [17]), whereas showing only the essential syntax constructs of a sequence diagram, to facilitate the mapping to the HTSs.

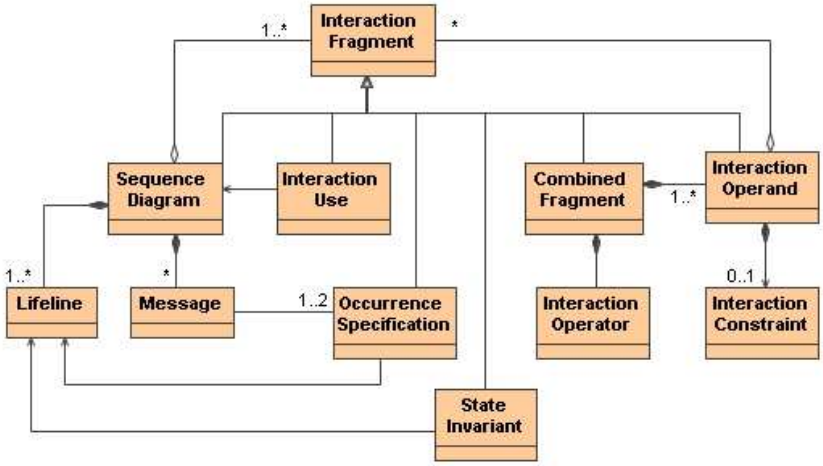

Fig. 1. UML 2 Sequence Diagram Representation UML Metamodel

In the metamodel, the syntax elements are represented as classes, shown as boxes, and relations among elements are represented as associations, shown as lines among classes in terms of class diagram. A hollow diamond on an association represents aggregation relationship (has-a), while a filled diamond represents a composition relationship (part-of). A triangle on an association represents a generalization between a superclass and its subclass. The numbers attached to an association are called multiplicities, which describe how many objects may exist in the association. A star denotes zero or more. If no multiplicity is present, a one-to-one relationship is implied.

A sequence diagram is a kind of InteractionFragment. It contains one or more Lifelines and zero or more Messages. A sequence diagram may have one or more InteractionFragments, which can be an InteractionUse, an OccurrenceSpecification, a StateInvariant, a CombinedFragment, and an InteractionOperand.

An InteractionUse is associated with the sequence diagram that it refers to. Both OccurrenceSpecification and StateInvariant are placed on a Lifeline, and have a unidirectional association with it. A Message can have one or two OccurrenceSpecifications.

A CombinedFragment contains an InteractionOperator and one or more InteractionOperands. An InteractionOperand may have an InteractionConstraint and can contain CombinedFragment and InteractionUse. The metamodel describes the structure of a sequence diagram. The detailed definitions will be presented in subsection C.

B. Running Example

In this paper, we use a simple ATM example (see Figure 2) to illustrate the syntax and semantics of UML 2 sequence diagram. This sequence diagram represents the withdraw cash scenario of an ATM system. We use UML modeling tool, called MagicDraw[1], to build the example, where Messages are numbered top-down.

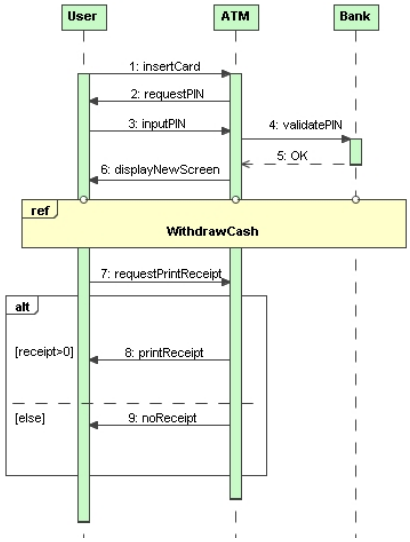

Fig. 2. Example of Sequence Diagram.

C. Basic Constructs of Sequence Diagram

A **Sequence diagram** describes an interaction which is a unit of behavior focusing on the observable exchange of messages between Objects (page 481 in [17]). The sequence diagram shown in Figure 2 shows an Interaction between objects in an "ATM" system.

A sequence diagram has two dimensions, where the vertical dimension represents time, and the horizontal dimension represents objects participating in the interaction (p585 [20]). The vertical dashed lines, called **Lifelines**, represent individual participants over a period of time in the Interaction (page 490 in [17]). Each Lifeline should have a name. In Figure 2, for example, "User", "ATM", and "Bank" name the three Lifelines.

The horizontal lines between Lifelines are called **Messages**. A Message is sent from the calling (source) Lifeline to the called (target) Lifeline. A Message can be a signal or an operation call. When a Message represents an operation call, the Message

[1]MagicDraw is a trademark of No Magic Inc.

is usually synchronous. After sending a Message, the source Lifeline is blocked until it receives a response from the target Lifeline [17], [20]. When a Message represents a signal, the Message is usually asynchronous. In Figure 2, the horizontal line named "insertCard" is a Message, it is from Lifeline "User" to Lifeline "ATM".

A Message has two ends. An intersection point between a Message and a Lifeline is called **OccurrenceSpecification** (OS). In Figure 2, Message "insertCard" ending on the Lifeline "User" is the sending OS and ending on Lifeline "ATM" is the receiving OS.

A Lifeline contains an ordered list of OSs, each of which models the occurrence of an event (page 435 in [20]). The semantics of an **Interaction** (e.g., sequence diagram) is defined in terms of trace which is a sequence of event occurrences within the execution of an interaction. The order of these OSs represents the time sequence in which the events occur [20]. Two OSs on different Lifelines are related if a message connect them: the sending OS executes before the receiving OS.

An interaction defines a set of allowed traces and a set of forbidden traces. All the traces in the allowed set are definitely consistent with the interaction definition. All the traces in the forbidden set are definitely inconsistent with the interaction definition (page 655 in [20]).

A **StateInvariant** is a runtime constraint on the Lifeline. It is evaluated before the next OS executes. If the constraint is true, then the trace is a valid trace, otherwise, the trace is an invalid trace [17].

D. Structured Control Constructs

UML 2 introduced structured control constructs, such as InteractionUse and CombinedFragment to represent more complex flow of control in a sequence diagram. These new features make sequence diagrams more expressive than UML 1 by allowing the representation of more complete interaction behavior.

InteractionUse allows one sequence diagram to refer to another sequence diagram. It copies the contents of the referred sequence diagram. When an InteractionUse executes, the effect is the same as executing the referenced sequence diagram (page 412 in [20]).

CombinedFragments represent different types of flows of control, such as concurrency, choice, and loop. A CombinedFragment consists of an InteractionOperator and one or more InteractionOperands. An InteractionOperand is an InteractionFragment and may contain an InteractionConstraint, which is a boolean expression.

InteractionOperators as defined in the most current OMG Superstructure [17] are listed below.

- **Alternatives**: has two or more operands, and at most one operand is chosen to execute.
- **Option**: has one operand with an InteractionConstraint, and the option operand is executed if the InteractionConstraint is true.
- **Break**: has one operand with or without an InteractionConstraint. If the constraint evaluates to true, the operand executes. Otherwise, the remainder of the enclosing InteractionFragment executes. If there is no InteractionConstraint, either the operand or the remainder of InteractionFragment executes.
- **Parallel**: has two or more operands. The execution of OSs in different operands may be interleaved in any order, but must be consistent with the ordering imposed by each operand separately.
- **Critical Region**: has one operand and the OSs on a single Lifeline must not be interleaved with any other OS in other InteractionFragments.
- **Loop**: has one operand with an InteractionConstraint. The operand will execute at least the minimum count and no more than the maximum count as long as the Interaction Constraint is true.

There are other InteractionOperators, such as Weak Sequencing, Strict Sequencing, Negative, Ignore, Consider and Assertion which are also defined in [17], [20].

An InteractionFragment is a structural piece of an Interaction which represents the body of an Interaction or a portion of an Interaction, such as a CombinedFragment and an InteractionUse. In Figure 2, both InteractionUse "WithdrawCash" and CombinedFragment "alt" are InteractionFragments. The CombinedFragments and InteractionUse define multiple independent subtraces of event occurrences. Events of subtraces will be combined in different ways based on the meaning of each InteractionOperator to make a single trace. The order in a single trace must adhere to the order of each subtrace, which is imposed by each InteractionOperand (page 509 in [20]). This is used to describe concurrency within interactions (page 655 in [20]).

III. Template Semantics

In this section, we give an overview of template semantics. Template semantics is a template-based approach to structure the operational semantics of behavioral specification notations. The goal of this approach is to make it easier to precisely document the semantics of notations by capturing notations' common semantics as predefined templates. Therefore, defining a notation's semantics can be reduced to providing an instantiation of the **template definitions** with a collection of **template parameter** values. The basic computation model of template semantics is a non-concurrent, **hierarchical transition system (HTS)**. Concurrency is supported by composition of HTSs.

A. Syntax of HTSs

As an extended state-machine, an HTS has a set of hierarchical control states, a set of transitions between control states, a set of events, and a set of typed variables. A transition is enabled by events or conditions. The execution of an enabled transition transforms an HTS from one state into another state, generates new events, and assigns new values to variables.

B. Semantics of HTSs

The template semantics of an HTS defines a specification's behavior in terms of a sequence of **snapshots**. A **snapshot** collects information about the current status of an HTS using snapshot elements. The snapshot elements capture an HTS's current states, current events, and current variable values and accumulate data about states, events, and variable values. These snapshot elements are used by different notations for the purpose of identifying a set of enabled transitions and recording the effect of the execution of an enabled transition.

A snapshot relation relates two snapshots if an HTS can move from one snapshot to a successor snapshot in a **step**. Template semantics defines two types of steps: a **micro-step** is the execution of a single transition per HTS, and a **macro-step** is a sequence of zero or more micro-steps. We use snapshot relation $N_{micro}(ss, \tau, ss')$ to characterize that taking enable transition τ in the current snapshot ss is an admissible step and its execution results in next snapshot ss'. A macro-step is initiated by a new set of inputs and is ended either when an HTS reaches a **stable snapshot**, in which no transition is enabled, or is ended by executing a single micro-step or an idle step.

Behavior that is common among notations is pre-defined by a set of template definitions, i.e., micro-step, macro-step, stable snapshot, **enabled transitions**, **reset**, and **apply**. Template definitions *enabled transitions* computes the set of transitions that are enabled, *apply* determines how the execution of an enabled transition affects the system, and *reset* incorporates new inputs into the current snapshot at the beginning of a macro-step. These template definitions are parameterized using the smaller, orthogonal template parameters. For example, parameters that specify *enabling states*, *enabling events*, and *enabling variable values* instantiate the template definition of *enabled transitions* to create a notation-specific function for determining which transitions are enabled in a given execution state.

C. Composition Operators

HTSs can be combined by composition operators to form a more complex, concurrent specification, expressed as a composition hierarchy of HTSs. Because composition operators are binary, the composition hierarchy is a binary tree. The template semantics of composition operators describes how multiple HTSs execute concurrently and how they communicate and synchronize with each other by exchanging events and data, and transferring control to one another.

Seven composition operators, (i.e., interleaving, parallel, environmental synchronization, rendezvous, sequence, choice, and interrupt) have been formally defined in our previous paper [14]. Each composition operator specifies how the components' snapshots change when the components take a collective step by representing the composite step relation as constraints on how to override the components' snapshot relations. Below, we review the semantics of two composition operators, interleaving and sequence, and the macro definitions that facilitate the specification of the transfer of control and exchanges of shared variables and events.

In defining the semantics of composition operators, we use vector notation $\vec{ss}$ and $\vec{\tau}$ to represent a snapshot tree and a transition tree, which match the composition hierarchy of HTSs. Predicate $comp1steps$ (Figure 3) captures the case in which one component takes a micro-step, and the other component's snapshots are simply updated to include the shared events and variable assignments generated by the executing component's transitions. Only component one executes its transitions $\vec{\tau}_1$, which is captured by setting component two's transitions $\vec{\tau}_2$ as empty. $update(\vec{ss}_2, \vec{\tau}_1, \vec{ss}'_2)$ defines the event-related elements in the next snapshot tree $\vec{ss}'_2$, which reflect the shared events generated and consumed by transitions $\vec{\tau}_1$ in the executing component. $communicate\_var((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2))$ defines the variable-related elements in next snapshots $\vec{ss}'_1$ and $\vec{ss}'_2$, so that variables shared among multiple snapshots all reflect

$$comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \equiv$$

$$\left[\begin{array}{ll} & N^1_{micro}(\overrightarrow{ss}_1, \overrightarrow{\tau}_1, \overrightarrow{ss}'_1) \wedge \overrightarrow{\tau}_2 = \emptyset \\ \wedge & update(\overrightarrow{ss}_2, \overrightarrow{\tau}_1, \overrightarrow{ss}'_2) \\ \wedge & communicate\_vars((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \emptyset), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \end{array}\right]$$

Fig. 3. Predicate for component 1 taking a step

$$N_{interleaving}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \equiv$$

$$\left[\begin{array}{ll} & comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \\ \vee & \\ & comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss}'_2, \overrightarrow{ss}'_1)) \end{array}\right]$$

Fig. 4. Semantics of Interleaving

the assignments made by all transitions executing in both components, and so that the final values of all shares variables are consistent.

The composite micro-step relation for the interleaving operator allows either component, but not both, to take a step even if both are enabled (Figure 4).

In sequence composition (Figure 5), two components execute in a sequential manner in three stages. In the first stage, component one executes and the shared variables of component two are updated (conjunct 1). The macro definition $basic\_states$, identifies which current states are basic states, in the test of whether component one has terminated. In the second stage, component one has reached its final states, control transfers to component two, component two starts to takes a step, and the component one's state-related snapshot elements are emptied, so that component one can no longer execute (conjunct 2). In the third stage, component two continues to execute and, for consistency, the snapshots of component one are updated (conjunct 3).

D. Template Semantics for Notations

Template semantics has been employed to document the semantics of many notations, such as UML stateMachines [21] basic transition systems, CSP, CCS, basic LOTOS, a variety of statecharts notations, a subset of SDL88, SCR, and Petri Nets [16]. The template-semantics description for a notation is an instantiation of the template parameters, and either mapping

$$N_{seq}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \equiv$$

$$\left[\begin{array}{ll} \wedge & basic\_states(\overrightarrow{ss}_1.CS) \neq S^F_1 \\ & comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss}'_1, \overrightarrow{ss}'_2)) \end{array}\right]$$

$$\vee \left[\begin{array}{ll} \wedge & basic\_states(\overrightarrow{ss}_1.CS) = S^F_1 \ \wedge \ \overrightarrow{ss}_1.CS \neq \emptyset \\ & comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1 \mid^{CS}_{\emptyset}), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss}'_2, \overrightarrow{ss}'_1)) \end{array}\right]$$

$$\vee \left[\begin{array}{ll} \wedge & \overrightarrow{ss}_1.CS = \emptyset \\ & comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss}'_2, \overrightarrow{ss}'_1)) \end{array}\right]$$

Fig. 5. Semantics of Sequence

the notation's composition operators to the pre-defined composition operators or defining new composition operators using our operators as a guide.

IV. Syntactic Mapping from UML 2 Sequence Diagram to HTS

As a first step to formalizing sequence diagrams, we need to represent a sequence diagram in terms of template semantics's computation model (HTS) and composition operators. A sequence diagram is mapped to a composition of CHTSs via rendezvous synchronization operators and interleaving operators with a specialized scheduling module. We transform a Lifeline into a CHTS (a set of HTSs composed via various interactionOperators). We map Messages between Lifelines to events communicated among CHTSs. A Lifeline is partitioned into a set of smaller pieces, which correspond to HTSs respectively.

To facilitate the mapping, we partition a sequence diagram into a set of maximal sequence fragments, which are adapted from the Maximal Independent Set in [5].

Definition 1 A **maximal sequence fragment** is a region of a sequence diagram, which contains a maximal sequence of consecutive Messages, and their associated Lifelines. It neither contains a CombinedFragment (except Critical Region) nor crosses the CombinedFragment's border, which consists of the solid-outline rectangle enclosing the CombinedFragment and the inside dashed horizontal lines separating the InteractionOperands. An InteractionConstraint together with the Message right below it are included in the same maximal sequence fragment.

If there is no CombinedFragment in a sequence diagram, then the sequence diagram only has one maximal sequence fragment. Otherwise, a maximal sequence fragment can be constructed by the following rules:

- The maximal sequence fragment can be the fragment from the sequence diagram's first Message to the upper border of the sequence diagram's first CombinedFragment.
- In a CombinedFragment, each operand that contains no other CombinedFragment is a maximal sequence fragment, which also includes the InteractionConstraint associated with the operand.
- Between any two adjacent CombinedFragments/InteractionUse, there is a maximal sequence fragment.
- The maximal sequence fragment is the fragment from the lower border of the sequence diagram's last CombinedFragment to the sequence diagram's last Message.
- An InteractionUse separates any constructed maximal sequence fragment using the above rules into two maximal sequence fragments.
- If an InteractionOperand contains a CombinedFragment, we apply the above rules recursively.

In Figure 1, there are four maximal sequence fragments. The first one contains the portion from Message 1 to Message 6. Message 7 constitutes the second maximal sequence fragment. The two operands of the CombinedFragments are the other two maximal sequence fragments, respectively.

Definition 2 (Maximal Block) A **maximal block** is the projection of a maximal sequence fragment on a single Lifeline. By projection, Messages are mapped to events, InteractionConstraints are mapped to conditions of transitions, where involved variables are variables.

In order to map a maximal block to an HTS, we use control states to represent intervals between two adjacent OSs. We add one control state, called initial state, before the first OS and another control state, called final state after the last OS for each maximal block.

Now we map a maximal block to an HTS as a 7-tuple $< S, S^I, S^F, E, V, V^I, T >$, where

- S is a finite set of control states.
- S^I describes an initial state.
- S^F describes a final state. If there is no OS in maximal block, the initial state and final state are unified.
- E is a finite set of events.
- V is a finite set of variables.
- V^I describes a set of all possible initial values of the variables in V.
- T is a finite set of transitions, which can be a sending OS or a receiving OS. The condition of transition includes Interaction Constraint of Combined Fragment.

The maximal block does not have any state hierarchy (S^H). A sending OS is the generation of an event in a transition, whereas a receiving OS is a triggering event. StateInvariants are the conditions of transitions.

A Lifeline consists of a set of maximal blocks. If maximal blocks are enclosed in InteractionOperands of the same CombinedFragment respectively, we compose them to a **Combined Maximal Block** using the CombinedFragment's InteractionOperators. For InteractionUse, we represent the set of maximal blocks that are enclosed in the referred sequence diagram as a combined maximal block as well. All the combined maximal blocks and maximal blocks are combined in a sequential manner. An Interaction is a composition of its lifelines, each of which is represented by a CHTS. Final states are introduced to represent the termination of maximal blocks. After each block reaches its final states, control is transferred to its subsequent block. In this way, we can represent the sequence diagram's termination of a sequence of messages.

V. Formalizing the Semantics of Sequence Diagram

Sequence diagram's semantics, provided by OMG, is defined in terms of traces of event occurrences, where the partial order of these events must be consistent with the ordering of the events imposed by each Lifeline. Therefore, the semantics of a sequence diagram is largely determined by its Lifelines. In this section, we formalize the semantics of a Lifeline in terms of template parameter values and micro-step definitions of composition operators. The composition of Lifelines forms a sequence diagram, which can be described using rendezvous and extended interleaving composition operators.

A. Template Parameters of Maximal Block

In our syntactic mapping, a lifeline is partitioned into a set of maximal blocks and combined maximal blocks that are composed via sequence operators. A step in a maximal block is

Template Parameter	Maximal Block
$reset\_CS(ss, I)$	$ss.CS$
$next\_CS(ss, \tau, CS')$	$CS' = dest(\tau)$
$en\_states(ss, \tau)$	$src(\tau) \subseteq ss.CS$
$reset\_IE(ss, I)$	I
$next\_IE(ss, \tau, IE')$	$IE' = ss.IE \cup gen(\tau)$
$en\_events(ss, \tau)$	$trig(I) \subseteq ss.IE$
$reset\_AV(ss, I)$	$assign(ss.AV, I.var)$
$next\_AV(ss, \tau, AV')$	$AV' = assign(ss.AV, eval(ss.AV, asn(\tau)))$
$en\_cond(ss, \tau)$	$ss.AV \models cond(\tau)$
$macro\_semantics$	$simple\ nondiligent$

Fig. 6. Template parameters for maximal block

the execution of an OS, thus, the semantics of a maximal block (i.e., a trace of OSs) can be defined as a sequence of steps of an HTS in template semantics. We instantiate template definitions with specific template parameter values to express maximal block's step semantics, illustrated in Figure 6.

$reset\_CS$ specifies that the current state stays the same at the start of each macro-step. The $next\_CS$ template parameters specifies that after the execution of transition τ, the next state is the destination state of the transition. en_states specifies if the current state is the source state of τ, τ can be enabled.

$reset\_IE$ specifies current events value, which is updated by the input event at the start of a macro-step. $next\_IE$ specifies after the execution of transition τ, the current event is updated to include the generated event by τ. en_events specifies whether snapshot's event-relate elements can enable transition τ.

At the start of a macro-step, $reset\_AV(ss, I)$ updates the current variable values using the input value, such as the arguments of an input event, $next\_AV$ specifies after the execution of a transition τ, the current variable is updated by the action of τ, $en\_cond(ss, \tau)$ specifies whether the condition of transition τevaluates to true.

We use simple macro-step for sequence diagram semantics, which means an HTS takes at most one micro-step or an idle step. The control states after the receiving OSs have no state-change, they can be combined when synthesizing the state machine later.

B. Compositional Semantics of Combined Maximal Block

A combined maximal block is a composition of maximal blocks using InteractionOperators, such as "alt", "opt", "break", "par", "critical region", and "loop". In this subsection, we extend template semantics compositional operators to define these InteractionOperators.

*1) **Alternatives:*** The InteractionOperator "**alt**" chooses at most one of its operands to execute. Each operand may have an explicit or an "else" constraint. The else constraint is the negation of the disjunction of all explicit constraints in the enclosing combined maximal block. An implicit constraint always evaluates to true (page 468 in [17]). The chosen operand's constraint must evaluate to true. If none of the operands has a constraint that evaluates to true, we add an empty "else" operand. We adapt the choice composition operator of template semantics to specify "alt" operator. We define "alt" as a binary composition operator, and its multiple operands can be composed using multiple binary "alt" operators.

$$
\begin{aligned}
&N_{alt}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \equiv \\
&\left[\begin{array}{l}
\left(\begin{array}{l}
\overrightarrow{ss_1.CS} \subseteq S_1^I \ \wedge \ \overrightarrow{ss_2.CS} \subseteq S_2^I \ \wedge \\
\overrightarrow{ss_1.CS} \neq \emptyset \ \wedge \ \overrightarrow{ss_2.CS} \neq \emptyset
\end{array}\right) \\
\wedge \\
\left(\begin{array}{l}
comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2 \mid_{S_2^F}^{CS}), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \\
\vee \\
comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1 \mid_{S_1^F}^{CS}), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}'))
\end{array}\right)
\end{array}\right] \\
\vee &\left[\begin{array}{l}
\left(\begin{array}{l}
basic\_states(\overrightarrow{ss}_2.CS) = S_2^F \\
\wedge \\
comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}'))
\end{array}\right) \\
\vee \\
\left(\begin{array}{l}
basic\_states(\overrightarrow{ss}_1.CS) = S_1^F \\
\wedge \\
comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}'))
\end{array}\right)
\end{array}\right]
\end{aligned}
$$

Fig. 9. Semantics of Alternatives

$$
\begin{aligned}
&N_{opt}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}') \equiv \\
&\quad [comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}'))]
\end{aligned}
$$

Fig. 10. Semantics of Option

Figure 9 shows the definition of "alt". Initially, one of the two operands can be nondeterministically chosen to execute if both of their constraints evaluate to true. Otherwise, only the operand, whose constraint evaluates to true, executes. After this choice is made, the combine maximal block behaves only like the chosen operand. The other operand is prevented to execute by emptying its state-related snapshot elements. For example, in Figure 2, the User requests ATM to print a receipt. The ATM prints receipt for the User if it still has receipt paper stored or displays no receipt on screen.

*2) **Option:*** The interactionOperator "**opt**" represents a choice of behavior that either the (sole) operand happens or nothing happens. "Opt" is semantically equivalent to "alt" with two operands, where one operand is the "opt" operand with non-empty content and the other operand is empty (page 468 in [17]).

Figure 10 shows the definition of "opt". $\overrightarrow{ss}_2$ is empty as it represents the snapshot tree of an empty operand. Figure 7 shows an example of withdraw cash. Before cash is withdrew, the Bank checks the balance of the User's account. If the balance is more than the withdraw amount, ATM deducts the User's debit account, otherwise, cash is withdrew from the User's credit account.

*3) **Break:*** We denote a sequence of all the combined maximal blocks or maximal blocks before "break" as **preBlock**. All the maximal blocks after "break" is denoted as **postBlock**. $N_{alt}((\overrightarrow{ss}_{break}, \overrightarrow{ss}_{post}), (\overrightarrow{\tau}_{break}, \overrightarrow{\tau}_{post}), (\overrightarrow{ss}'_{break}, \overrightarrow{ss}'_{post})$shows the definition of "break" in terms of "alt" (see Figure 9). If the break operand is chosen, the operand executes. Otherwise, postBlock executes. If there is no constraint, either the break

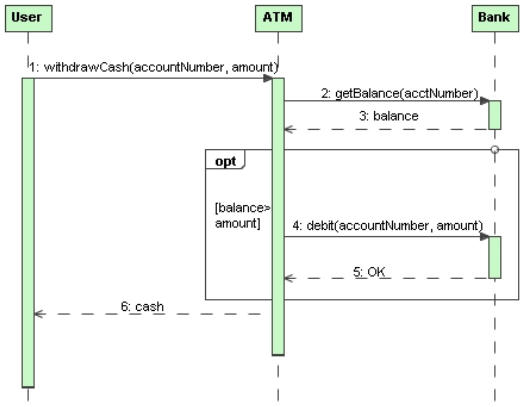

Fig. 7. Example of Option

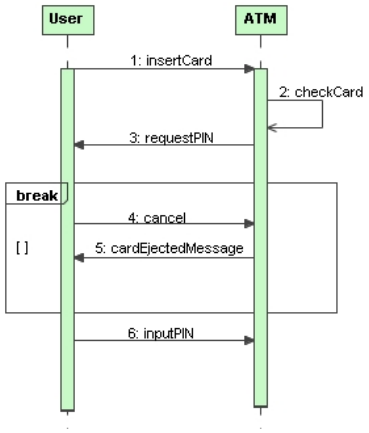

Fig. 8. Example of Break

operand or the postBlock is nondeterministically chosen to execute. An example illustrates in Figure 8, shows after the User inserts a card and the ATM validates the card, the ATM requests a PIN from the User. If the User decides to cancel the transition, the ATM will eject the card and the the rest of the scenario after the break operand is ignored. Otherwise, the break operand is ignored and the User inputs the PIN and continue his/her transition.

*4) **Parallel**:* The interactionOperator "**par**" represents the OSs of the different operands can be interleaved as long as the ordering imposed by each operand is preserved (page 468 in [17]). We use the interleaving composition operator to define the "par" operator (See Figure 4). Interleaving operator allows all possible orders of OSs that adhere to each operand to be explored. For example, in Figure 12, on Server Lifeline, each operand imposes an order to its OSs, where OS of Message 1 is received before OS of Message 2, and OS of Message 3 is received before OS of Message 4. The interleaving of the two operands defines the following traces of OSs, $(1,2,3,4)$, $(1,3,2,4)$, $(1,3,4,2)$, $(3,1,2,4)$, $(3,1,4,2)$, and $(3,4,1,2)$, are admissible.

*5) **Critical Region**:* The InteractionOperator "**critical**" restricts OSs within its sole operand from being interleaved with other OSs that are enclosed in the parallel operands. Figure 11 shows the definition of "par", whose operands may contain Critical Regions. $s\vec{s}_1$ and $s\vec{s}_2$ represent the snapshot trees of the two operands of "par". We assume there is at most one Critical Region within a single operand of "par". T_i is the set of the transitions within Critical Region in "par" operand i, where T_i^I is the initial transition, T_i^F is the final transition, and $i \in \{1,2\}$. In case 1, Critical Region within one parallel operand starts to execute, and the other parallel operand is put on hold by copying its current states to its auxiliary snapshot element CS_a and clearing its current states. In case 2, the operand continues to execute transition until reaching the final transition in the Critical Region. In case 3, when the executing operand has reached the final transition enclosed in the Critical Region, the value of CS_a is copied back to the other "par" operand's current states. In case 4, if neither "par" operand executes transitions in Critical Region, either "par" operand

$$
\begin{aligned}
&N_{parallel}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}'),\\
&(T_1, T_2), (T_1^I, T_2^I), (T_1^F, T_2^F)) \equiv\\
&\left[\begin{array}{l} \overrightarrow{\tau}_1 = T_1^I \\ \wedge\ comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2 \,|_{CS}^{CS_a}), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}' \,|_{\emptyset}^{CS})) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{\tau}_2 = T_2^I \\ \wedge\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1 \,|_{CS}^{CS_a}), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}' \,|_{\emptyset}^{CS})) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{ss_2}.CS = \emptyset \wedge \overrightarrow{\tau}_1 \subseteq T_1 \wedge \overrightarrow{\tau}_1 \neq T_1^F \\ \wedge\ comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{ss_1}.CS = \emptyset \wedge \overrightarrow{\tau}_2 \subseteq T_2 \wedge \overrightarrow{\tau}_2 \neq T_2^F \\ \wedge\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}')) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{\tau}_1 = T_1^F \\ \wedge\ comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2 \,|_{CS_a}^{CS}), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{\tau}_2 = T_2^F \\ \wedge\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1 \,|_{CS_a}^{CS}), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}')) \end{array}\right]\\
\vee &\left[\begin{array}{l} \overrightarrow{\tau}_1 \not\subseteq T_1 \wedge \overrightarrow{\tau}_2 \not\subseteq T_2 \\ \wedge \left[\begin{array}{l} comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \\ \vee\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}')) \end{array}\right] \end{array}\right]
\end{aligned}
$$

Fig. 11. Semantics of Parallel with Critical Region

can be chosen to execute. If there is no Critical Region in any operand, the two parallel operands are interleaved. In an example shown in Figure 13, two operands of parallel CombinedFragment are interleaved, however the order of the OSs enclosed in critical region operand must be kept. On the Lift Lifeline, OSs of Message 3, 4, and 5 must happen in a sequential manner, i.e., once Message 3 starts, Message 4 and 5 can not be interrupted by Message 1 or 2. In this way, the traces of OSs on Lift can only be $(1,2,3,4,5)$, $(1,3,4,5,2)$ and $(3,4,5,1,2)$ where $(3,4,5)$ must occur as a whole.

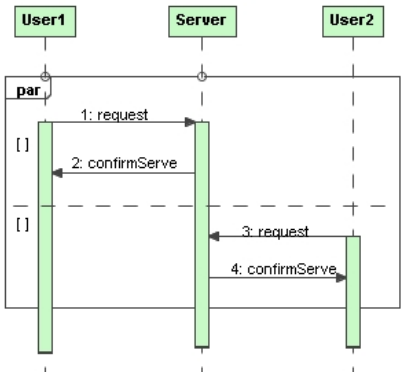

Fig. 12. Example of Parallel

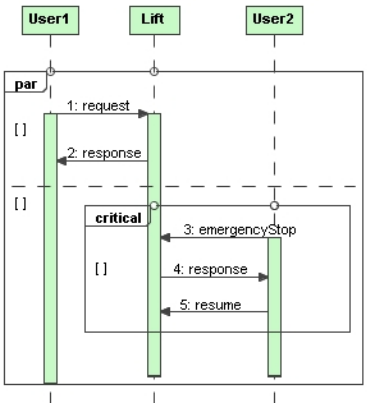

Fig. 13. Example of Critical Region

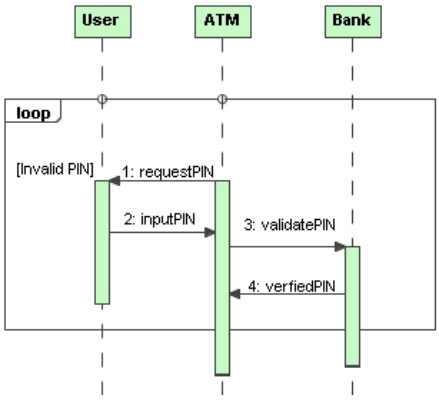

Fig. 14. Example of Loop

*6) **Loop**:* The InteractionOperator "**loop**" contains one operand with a constraint. It includes a lower bound, 'minint', and an upper bound, 'maxint', to set restriction on the number of iterations. A loop iterates at least the 'minint' number of times and at most the 'maxint' number of times. If the constraint evaluates to false after the minimum number of iterations, the loop will terminate (page 470 in [20]). Figure 15 represents the semantics of "loop". $count$ and $count'$ represents the loop iterations. $minint$ and $maxint$ are the lower and upper number of iterations of the loop, The default value of $minint$ is zero and the default value of $maxint$ is ∞. We denote the maximal block or combined maximal block after "loop" as **postBlock**. $s\vec{s}_1$ is the snapshot tree of loop operand and $s\vec{s}_2$ is the snapshot tree of the postBlock. In case 1, the loop has not iterated 'minint' number of times, the "loop" operand starts to execute. In case 2, the loop has iterated more than 'minint ' times but less than 'maxint' times, so the "loop" operand starts to execute if the conditions are satisfied. In case 3, the "loop" operand continues to execute until $s\vec{s}_1$ reaches its final states. At the beginning or when $s\vec{s}_1$ reaches its final states, if the loop has iterated 'maxint' times, or if the the loop has iterated more than 'minint' times and the condition of $s\vec{s}_1$ evaluates to false, the postBlock starts to executes and the current states of $s\vec{s}_1$ are cleared in case 4. In case 5, the postBlock continues to executes. Figure 14 shows an example of requesting PIN. The ATM requests PIN and the User inputs PIN for bank to validate, if the PIN is invalid, the ATM requests PIN again but not more than the maxint (e.g. 3) times until the PIN is verified.

*7) **Weak Sequencing**:* The InteractionOperator" **seq**" represents a weak sequencing between the behaviors of the operands. It is defined by the properties: 1). In the same operand, the ordering of the OSs are maintained. 2). On different lifelines from different operands, the OSs are interleaving. 3) On the same lifeline from different operands, the OSs come from the first operand happen before the OSs come from the second operand[17]. If there are more than one Lifeline in the sequence diagram, the semantics of Weak Sequencing is similar with the semantics of Parallel. We can use the interleaving composition operator to define the "seq" operator too.(See Figure 4). Two components in the interleaving operator semantics represent two snapshot trees from different lifelines in different operand. If only one Lifeline in the sequence diagram, the Weak

$$
\begin{aligned}
&N_{loop}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}'), (count, count'), \\
&(minint, maxint)) \equiv \\
&\left[\begin{array}{l} \left[\begin{array}{l} basic\_states(\overrightarrow{ss}_1.CS) = S_1^I \\ \vee\ basic\_states(\overrightarrow{ss}_1.CS) = S_1^F \end{array}\right] \\ \wedge\ (count < minint) \\ \wedge\ (count' = count + 1) \\ \wedge\ comp1steps_{uncond}((\overrightarrow{ss}_1\ |_{S_1^I}^{CS}, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right] \\
&\vee \\
&\left[\begin{array}{l} \left[\begin{array}{l} basic\_states(\overrightarrow{ss}_1.CS) = S_1^I \\ \vee\ basic\_states(\overrightarrow{ss}_1.CS) = S_1^F \end{array}\right] \\ \wedge\ (minint \leq count < maxint) \\ \wedge\ (count' = count + 1) \\ \wedge\ comp1steps((\overrightarrow{ss}_1\ |_{S_1^I}^{CS}, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right] \\
&\vee \\
&\left[\begin{array}{l} \left[\begin{array}{l} basic\_states(\overrightarrow{ss}_1.CS) \neq S_1^I \\ \vee\ basic\_states(\overrightarrow{ss}_1.CS) \neq S_1^F \end{array}\right] \\ \wedge\ \overrightarrow{ss}_1.CS \neq \emptyset \\ \wedge\ comp1steps((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right] \\
&\vee \\
&\left[\begin{array}{l} \left[\begin{array}{l} basic\_states(\overrightarrow{ss}_1.CS) = S_1^I \\ \vee\ basic\_states(\overrightarrow{ss}_1.CS) = S_1^F \end{array}\right] \\ \wedge\ [(maxint \leq count) \vee \neg eval(\overrightarrow{ss}_1.AV, \overrightarrow{\tau}_1.cond)] \\ \wedge\ minint \leq count \\ \wedge\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1\ |_{\emptyset}^{CS}), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}')) \end{array}\right] \\
&\vee \\
&\left[\begin{array}{l} \overrightarrow{ss}_1.CS = \emptyset \\ \wedge \\ comp1steps((\overrightarrow{ss}_2, \overrightarrow{ss}_1), (\overrightarrow{\tau}_2, \overrightarrow{\tau}_1), (\overrightarrow{ss_2}', \overrightarrow{ss_1}')) \end{array}\right]
\end{aligned}
$$

Fig. 15. Semantics of Loop

$$comp1steps_{uncond}((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \overrightarrow{\tau}_2), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \equiv$$

$$\left[\begin{array}{ll} & en\_states(\overrightarrow{ss}_1, \overrightarrow{\tau}_1) \wedge en\_events(\overrightarrow{ss}_1, \overrightarrow{\tau}_1) \\ \wedge & apply(\overrightarrow{ss}_1, \overrightarrow{\tau}_1, \overrightarrow{ss}_1') \wedge \overrightarrow{\tau}_2 = \emptyset \\ \wedge & update(\overrightarrow{ss}_2, \overrightarrow{\tau}_1, \overrightarrow{ss}_2') \\ \wedge & communicate\_vars((\overrightarrow{ss}_1, \overrightarrow{ss}_2), (\overrightarrow{\tau}_1, \emptyset), (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \end{array}\right]$$

Fig. 16. Predicate for component 1 taking a step without evaluating condition

Sequencing operand reduces to strict sequencing operand[17], we can use the semantics of sequence to represent it. (See Figure5).

C. Semantics of InteractionUse

In UML, InteractionUse is shown as a CombinedFragment symbol where the operator is called "**ref**" [17]. InteractionUse copies the contents of the referred sequence diagram to the "**ref**" operand. The "ref" operator is defined using template semantics compositional operator Sequence as shown in Figure 5. $N_{seq}((\overrightarrow{ss}_{pre}, \overrightarrow{ss}_{ref}), (\overrightarrow{\tau}_{pre}, \overrightarrow{\tau}_{ref}), (\overrightarrow{ss}'_{pre}, \overrightarrow{ss}'_{ref}))$ shows the definition of "ref" where operand pre is the preBlock before "ref".

There are some other composition operators, such as: Negative, Assertion, Coregion, Strict Sequence, Ignore/Consider, Part Decomposition and Continuation, which have not been defined in the paper. We believe their semantics can be represented by extending template semantics composition operators.

VI. Related work

There has been substantial related work on formalizing the semantics of sequence diagrams and message sequence charts. As message sequence charts (MSCs) have much influence on UML 2 sequence diagram, we include in this section the research efforts of providing the formal semantics of both notations.

Usually, the purpose to document a language's precise semantics is to facilitate the communication and understanding of such a language, and possibly as a step towards synthesizing behavior models and developing reasoning and verification tools. Uchitel et. al. define the semantics of MSCs in terms of labeled transition systems [22]. Because a basic MSC often models a scenario, which is a partial execution of a system, multiple charts need to be composed to capture the complete behavior. Those composition operators, i.e., sequential, choice and iteration operators, are formalized, and synchronous communication among entities participating in a scenario is assumed. In this way, they are able to synthesis a behavioral specification in the form of Finite Sequential Processes, which can be checked using the associate analyzer of labeled transition systems. Letichevsky et. al. define the semantics of MSC based on the theory of interaction of agents and environments [1]. MSCs, compared with UML 2 Sequence Diagrams, are less expressive in that they support only synchronous messages without conditions and a small set of compositions, not including interleaving and critical region, .

Working towards similar goals, Damas et. al. synthesize labeled transition system model from both positive and negative scenarios, expressed in MSC [4]. They adopt the semantics definitions from [22]. In addition, they generate temporal properties from scenarios. Lamsweerde et.al.[23] develop an approach to inferring system properties, in terms of temporal logic, from both positive and negative scenarios. A scenario, expressed as a MSC, is defined as a temporal sequence of interaction events among different agents.

Whittle and Schumann [24] propose a way to combine UML 1 sequence diagram models and to translate them into UML statecharts. This approach requests users to add pre-conditions and post-conditions of each message in terms of UML Object Constraint Language (OCL). The pre-conditions and post-conditions provide messages with annotations to refine their meanings. They only develop the synthesis of multiple scenarios but do not attempt to formally check them.

To model check MSCs, Alur et. al. formalize them using automata [3], [2]. Peled et. al. present an extension of high-level message sequence chart(HMSC), which is called high-level composition message sequence chart(HCMSC). [18], [6], [13]. They describe the semantics of HCMSCs in terms of ω-automata. Holzmann et al. [11] also create techniques and tools to analyze MSCs. They define semantics of MSCs by assigning a specific causal ordering among messages. The analysis checks for safety and liveness properties using graph theory.

Eichner et. al. introduce a compositional formal semantics of UML 2 sequence diagram using colored high-level Petri Nets [5]. The semantics can represent most of the InteractionOperators of sequence diagram. Their formalization is able to express both the sequential and concurrent behavior of a sequence diagram. Haugen et. al. present formal semantics of UML 2 sequence diagram by an approach named STAIR [9]. STAIR provides a trace-based representation of InteractionOperations, such as "alt", "neg", "par", and "seq".

VII. Conclusions

We have demonstrated in this paper that a sequence diagram can be formalized using template semantics. We build, for a sequence diagram, a metamodel, which can ease users' efforts to understand the structure of sequence diagrams. We partition a Lifeline into a set of maximal blocks, such that their semantics can be represented using a sequence of steps. We also extend composition operators of template semantics to provide a compositional semantics of sequence diagrams. We believe that this formalization provides a possibility for practitioners to precisely document requirements (e.g., scenarios of use cases) of high assurance systems. In addition, it will enable not only the synthesis of more succinct behavior models, such as state machines from multiple sequence diagrams, but also a means for tool builders to develop analysis tools to detect subtle errors in high assurance systems.

In the future, we plan to prove the commutative and associative properties of InteractionOperators, such as "alt" and "par", to justify the use of binary operators for composing multiple maximal blocks. We are also working towards building tools to synthesize a behavioral model from sequence diagrams, such that we can model check their desired properties.

References

[1] A.A.Letichevsky, J.V.Kapitonova, V.P.Kotlyarov, V.A.Volkov, A.A.Letichevsky Jr., and T.Weigert. Semantics of message sequence charts. *SDL 2005: Model Driven Systems Design, LNCS*, 3530, 2005.

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331, 2005.

[3] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory, LNCS*, volume 1664, pages 114 – 129, 1999.

[4] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31, 2005.

[5] Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. Compositional semantics for UML 2.0 sequence diagram using Petri Nets. *SDL 2005: Model Driven Systems Design, LNCS*, 3530:133–148, 2005.

[6] Elsa L. Gunter, Anca Muscholl, and Doron Peled. Compositional message sequence charts. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, volume 2031, pages 496 – 511, 2001.

[7] David Harel et al. On the formal semantics of statecharts. In *Proceedings of Logic in Computer Science*, pages 54–64, 1987.

[8] David Harel and Amnon Naamad. The Statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[9] Oystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stolen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4, November.

[10] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, UK, 1985.

[11] Gerard J. Holzmann, Doron Peled, and Margaret H. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 2(1):86–95, 1997.

[12] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[13] Anca Muscholl and Doron Peled. Deciding properties of message sequence charts. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure, LNCS*, volume 1378, pages 226 – 242, 1998.

[14] Jianwei Niu. *Template Semantics: A Parameterized Approach to Semantics-Based Model Compilation*. PhD thesis, University of Waterloo, May 2005.

[15] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866–882, October 2003.

[16] Jianwei Niu, Joanne M. Atlee, and Nancy A. Day. Understanding and comparing model-based specification notations. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE)*, pages 188–199, 2003.

[17] Object Management Group. Unified Modeling Language: Superstructure v2.1.2.

[18] Doron Peled. Specification and verification using message sequence charts. In *Formal Techniques for Distributed System Development, FORTE/PSTV*, pages 139 – 154, 2000.

[19] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[20] James Rumbaugh, Ivar Jacobon, and Grady Booch. *The Unified Modeling Language Reference Manual Second Edition*. Addison-Wesley, July 2004.

[21] Ali Taleghani and Joanne M. Atlee. Semantic variations among UML statemachines. In *Proceedings of the 9th International Conference Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2006.

[22] Sebastian Uchitel, Jeff Kramer, and Jeff Magge. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29, February.

[23] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios, 1998.

[24] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios, 2000.