A Logical Framework for Sequence Diagram with Combined Fragments

Hui Shen, Mark Robinson and Jianwei Niu
Department of Computer Science
University of Texas at San Antonio
One UTSA Circle, San Antonio, Texas, USA
Email: {hshen, mrobinso, niu}@cs.utsa.edu


Abstract—Graphical representations of scenarios, such as UML Sequence Diagrams and Message Sequence Charts, serve as a well-accepted means for modeling the interactions among software systems and their environment through the exchange of messages. The Combined Fragments of UML Sequence Diagram permit various types of control flow among messages (e.g., interleaving and branching) to express an aggregation of multiple traces encompassing complex and concurrent behaviors. However, Combined Fragments increase the difficulty of Sequence Diagram comprehension and analysis. This paper introduces an approach to formalizing the semantics of Sequence Diagrams with Combined Fragments in terms of Linear Temporal Logic (LTL) templates. In all the templates, different semantic aspects are expressed as separate, yet simple LTL formulas that can be composed to define the semantics of basic Sequence Diagram, all the Combined Fragments, and nested Combined Fragments. Moreover, the formalization enables us to leverage the analytical powers of automated decision procedures for LTL formulas to determine if a collection of Sequence Diagrams is consistent, safe, and deadlock-free.

Keywords Modeling – Formal Analysis – Sequence Diagram – Concurrency & Communication – Temporal Logic

I. INTRODUCTION

The software development community is adopting model-driven engineering as a viable practice to improve the productivity and quality of software systems. Scenario-based models have been widely employed for the description of interactions among environmental actors (e.g., other software packages or human beings) and the components of the software systems. UML Sequence Diagrams, which graphically depict scenarios, serve as well-accepted and intuitive media among software practitioners and tool builders. UML 2 adds many major structural control constructs to the Sequence Diagram, including Combined Fragments and Interaction Use, to allow multiple, complex scenarios to be aggregated in a single Sequence Diagram.

Combined Fragments permit different types of control flow, such as interleaving and branching, for presenting complex and concurrent behaviors, increasing a Sequence Diagram’s expressiveness. The semantics of Sequence Diagram with Combined Fragments is described in terms of sets of valid and invalid traces by OMG. However, it is not formally defined how to derive the traces compared to their precise syntax descriptions [32]. A formal semantics of Sequence Diagrams is required to facilitate a user to interpret and analyze Sequence Diagrams.

Next, Combined Fragments can be nested to provide more combinations of control flows. For example, if a Combined Fragment presenting branching behavior is nested within a Combined Fragment presenting iteration behavior, different choices may be made in different iterations. Some Combined Fragments need to be nested within others to make them more significant. For example, for a Combined Fragment which defines a critical region, it may be nested within another Combined Fragment representing interleaving control flows. Nested Combined Fragments make it difficult to comprehend what traces can be derived to express possible system executions.

Further, software practitioners can construct multiple Sequence Diagrams that are complementary perspectives of a single system. Some Sequence Diagrams represent the possible system executions while others represent desired properties. Determining that these Sequence Diagrams provide a consistent specification can therefore be extremely difficult.

To address these issues and gain better theoretical understanding of a Sequence Diagram, we have developed a logical framework to represent its semantics in Linear Temporal Logic (LTL) [34], as LTL is a natural choice for specifying traces. We identify a collection of separate semantic aspects for a Sequence Diagram and for Combined Fragments respectively; these aspects are orthogonal and independent, in the sense that they do not constrain each other. We provide an LTL definition to represent each separate aspect to conquer the complexity of formalizing Sequence Diagram with Combined Fragments as a whole. The semantics that is common to the 12 Combined Fragments is captured as an LTL template, which is a conjunction of simpler definitions. The specifics of each Combined Fragment can be expressed as additional constraints, conjuncted to the common template to form a complete semantic definition. Nested Combined Fragments may also be represented as conjunctions of LTL definitions. As UML leaves many semantic variation points to be defined by the users, we believe the LTL definitions provided by our framework can be largely reused to formalize customizable semantics.
Working towards similar goals, several work has been proposed using different languages, including temporal logic [22], [24], automata [17], [19], [21], Petri nets (colored Petri nets) [14], [15], PROMELA [27], and template semantics [36], to provide a formal semantics for scenario-based notations, including UML 2 Sequence Diagrams and live sequence charts (LSC). Micskei and Waeselynck [28] survey and categorize 13 approaches of UML Sequence Diagram semantics. As one of the earliest approaches, Storrle [37] proposed a trace-based semantics of UML 2 Sequence Diagram, introducing the semantics of all 12 Combined Fragments. Motivated by analyzing scenarios based requirements, Kugler et al. [22] and Kumar et al. [24] have described the semantics of LSC using temporal logic. LSC is an extension of message sequence charts but distinguishes between behaviors which may happen from those which must happen. Their approach focuses on synchronous communication among objects, which can be applied to UML Sequence Diagrams with synchronous Messages. To support the Interaction Operators of Combined Fragments of UML 2, especially assert and negate, Harel and Maoz [19] propose a Modal Sequence Diagram (MSD), which is an extension of UML 2 Sequence Diagram based on the universal/existential concepts of Live Sequence Charts. Their approach increases the expressive power of Sequence Diagrams to specifying liveness and safety properties. They mainly consider synchronous Messages and Interaction Fragments are combined using Strict Sequencing. Grosu and Smolka [17] propose a formal semantics of UML 2 Sequence Diagrams based on the observation of positive and negative Sequence Diagrams. The positive and negative Sequence Diagrams represent liveness and safety properties respectively using Büchi automata. Their refinement of Sequence Diagrams provides multiple control flows as Combined Fragments.

Most of work does not cover the semantics of all the Combined Fragments, in particular, nested Combined Fragments, Interaction Constraints, and both synchronous and asynchronous Messages. Our logical framework provides a complete semantics description for all the Combined Fragments, as well as discussing the semantic differences between using Sequence Diagrams to represent possible system executions and to represent desired properties. One of the key benefits of formally modeling Sequence Diagrams using LTL is the ability to leverage the analytical power of automated analysis tools to reason about them.

Our approach bridges the gap between intuitive Sequence Diagrams and formal methods, increasing the accessibility of formal verification techniques to practitioners. To evaluate this hypothesis, we formally describe Sequence Diagrams with Combined Fragments in terms of NuSMV [8] modules as well. Using the LTL templates, we translate the Assertion and Negative Combined Fragments, representing mandatory and forbidden behaviors respectively, into LTL specifications to express consistency and safety properties of a system. The model checking mechanism can explore all possible traces specified in the Sequence Diagram, verifying if these properties are satisfied. Thus we can verify that a set of Sequence Diagrams is consistent and safe without requiring users to specify the LTL properties directly. We have developed a proof-of-concept tool to implement these techniques and generate Sequence Diagram visualizations from NuSMV counterexamples to ease user efforts to locate a property violation. We evaluate our approach by analyzing two design examples taken from an insurance industry software application.

The main contribution of our research is four-fold:

- We develop a logical framework to formalize the semantics of all the Combined Fragments, including nested Combined Fragments and Interaction Constraints, and both synchronous and asynchronous Messages. In the framework, different semantic aspects are expressed as separate, yet simple LTL formulas that can be composed to define the semantics of a Sequence Diagram.
- Additionally, we present how to generate safety and consistency properties as LTL formulas from Sequence Diagrams with Negative and Assertion Combined Fragments respectively.
- For a collection of Sequence Diagrams, we translate the Sequence Diagrams expressing possible executions into NuSMV models and transform other Sequence Diagrams expressing invalid and mandatory behaviors into LTL formulas, to verify the collection of Sequence Diagrams is safe and consistent.
- We develop a tool suite to implement the above techniques and visualize the NuSMV counterexamples to ease user efforts to locate the violation.

The rest of the paper is structured as follows. Section II summarizes the syntax and semantics of Sequence Diagrams. Section III presents the deconstructions of Sequence Diagram to facilitate the semantic definition. Sections IV and V describe the LTL templates to represent the semantics of Sequence Diagrams with Combined Fragments. Section VI discusses the generation of the LTL safety and consistency properties from Negative and Assertion Combined Fragments. Section VII describes the formal representation of Sequence Diagrams with Combined Fragments in terms of NuSMV modules. Section VIII introduces our framework for automated analysis of Sequence Diagrams and evaluates our approach via a case study. Section IX presents related work. We conclude with Section X.

II. UML 2 SEQUENCE DIAGRAM

In this section, we outline the syntax and semantics of a Sequence Diagram with Combined Fragments provided by OMG [32]. We begin with the basic Sequence Diagram, then discuss the structured control constructs, including Combined Fragments and Interaction Use.

A. Basic Sequence Diagram

We refer to a Sequence Diagram without Combined Fragments as a basic Sequence Diagram (see figure 1a for an example with annotated syntactic constructs). A Lifeline is a vertical line representing a participating object. A horizontal line between Lifelines is a Message. Each Message is sent
from its source Lifeline to its target Lifeline and has two endpoints, e.g., \( m1 \) is a Message sent from Lifeline \( L1 \) to Lifeline \( L2 \) in figure 1a. Each endpoint is an intersection with a Lifeline and is called an Occurrence Specification (OS), denoting a sending or receiving event within a certain context, i.e., a Sequence Diagram. OSs can also be the beginning or end of an Execution Specification, indicating the period during which a participant performs a behavior within a Lifeline, which is represented as a thin rectangle on the Lifeline.

The semantics of a basic Sequence Diagram is defined by a set of traces. A trace is a sequence of OSs expressing Message exchange among multiple Lifelines. We identify four orthogonal semantic aspects, each of which is expressed in terms of the execution order of concerned OSs, must be considered for the basic Sequence Diagram [28], [32]

1. On each Lifeline, OSs execute in their graphical order.
2. Each OS can execute only once, i.e., each OS is unique within a Sequence Diagram.
3. For a single Message, the sending OS must take place before the receiving OS does.
4. In a Sequence Diagram, only one object can execute an OS at a time, i.e., OSs on different Lifelines are interleaved.

Consider again figure 1a. OS \( r2 \) can not happen until OS \( r1 \) executes on Lifeline \( L2 \), which is prescribed by semantic aspect 1. All six OSs are uniquely defined, which is prescribed by semantic aspect 2. For Message \( m1 \), OS \( r1 \) can not happen until OS \( s1 \) executes, which is imposed by semantic aspect 3. OS \( s1 \) and \( s2 \) can not happen at the same time, which is imposed by semantic aspect 4.

Messages are of two types: asynchronous and synchronous. The source Lifeline can continue to send or receive other Messages after an asynchronous Message is sent. If a synchronous Message is sent, the source Lifeline blocks until it receives a response from the target Lifeline [32].

B. Combined Fragments

Both Combined Fragments and Interaction Use are structured control constructs introduced in UML 2. A Combined Fragment (CF) is a solid-outline rectangle, which consists of an Interaction Operator and one or more Interaction Operands. Figure 1b shows example CFs with annotated syntactic constructs. A CF can enclose all, or part of, Lifelines in a Sequence Diagram. The Interaction Operands are separated by dashed horizontal lines. The Interaction Operator is shown in a pentagon in the upper left corner of the rectangle. OSs, CFs, and Interaction Operands are collectively called Interaction Fragments. An Interaction Operand may contain a boolean expression which is called an Interaction Constraint or Constraint. An Interaction Constraint is shown in a square bracket covering the Lifeline where the first OS will happen. The CFs can be classified by the number of their Interaction Operands. Alternatives, Parallel, Weak Sequencing and Strict Sequencing contain multiple Operands. Option, Break, Critical Region, Loop, Assertion, Negative, Consider, and Ignore contain a single Operand. The example in figure 1b contains two CFs: a Parallel with two Operands and a Critical Region with a single Operand.

An Interaction Use construct allows one Sequence Diagram to refer to another Sequence Diagram. The referring Sequence Diagram copies the contents of the referenced Sequence Diagram.

The semantics of the \( seq \) Sequence Diagram with CFs is defined by two sets of traces, one containing a set of valid traces, denoted as \( Val(seq) \), and the other containing a set of invalid traces, denoted as \( Inval(seq) \). The intersection of these two sets is empty, i.e., \( Val(seq) \cap Inval(seq) = \emptyset \). Traces specified by a Sequence Diagram without a Negative CF are considered as valid traces. An empty trace is a valid trace. Invalid traces are defined by a Negative CF. Traces that are not specified as either valid or invalid are called inconclusive traces, denoted as \( Incon(seq) \). An Assertion specifies the set of mandatory traces in the sense that any trace that is not consistent with the traces of it is invalid, which is denoted as \( Mand(seq) \).

Along a Lifeline, OSs that are not contained in the CFs, are ordered sequentially. The order of OSs within a CF’s Operand which does not contain other CFs in it is retained if its Constraint evaluates to True. A CF may alter the order of OSs in its different Operands. We first identify three independent
C. Interaction Operator

The execution of OSs enclosed in a CF is determined by its Interaction Operator, which is summarized as follows:

- **Alternatives**: one of the Operands whose Interaction Constraints evaluate to *True* is nondeterministically chosen to execute.
- **Option**: its sole Operand executes if the Interaction Constraint is *True*.
- **Break**: its sole Operand executes if the Interaction Constraint evaluates to *True*. Otherwise, the remainder of the enclosing Interaction Fragment executes.
- **Parallel**: the OSs on a Lifeline within different Operands may be interleaved, but the ordering imposed by each Operand must be maintained separately.
- **Critical Region**: the OSs on a Lifeline within its sole Operand must be maintained separately.
- **Loop**: its sole Operand will execute for at least the minimum count (lower bound) and no more than the maximum count (upper bound) as long as the Interaction Constraint is *True*.
- **Assertion**: the OSs on a Lifeline within its sole Operand must occur immediately after the preceding OSs.
- **Negative**: its Operand represents forbidden traces.
- **Strict Sequencing**: in any Operand except the first one, OSs cannot execute until the previous Operand completes.
- **Weak Sequencing**: on a Lifeline, the OSs within an Operand cannot execute until the OSs in the previous Operand complete, the OSs from *different Operands on different Lifelines* may take place in any order (cf. Strict Sequencing).
- **Consider**: any message types other than what is specified within the CF is ignored.
- **Ignore**: the specified messages types are ignored within the CF.
- **Coregion**: the contained OSs and CFs on a Lifeline are interleaved.
- **General Ordering** imposes an order between two unrelated OSs on different Lifelines.

III. SEQUENCE DIAGRAM DECONSTRUCTION

In this section, we present the formal definitions of a Sequence Diagram. First, we give a textual representation of a Sequence Diagram. Then, we deconstruct a Sequence Diagram and CFs into fine-grained syntactic constructs to facilitate the semantic description of Sequence Diagram, in particular, Weak Sequencing among OSs and CFs.

A. Definition of Syntactic Constructs

A Sequence Diagram consists of a set of Lifelines and a set of Messages. A Message is the specification of an occurrence of a message type within the Sequence Diagram, while a message type is the signature of communications from one Lifeline to another. Each Message is uniquely defined by its sending OS and receiving OS, each of which is associated with a location of a Lifeline. Within the Sequence Diagram, an OS represents an occurrence of an event. The textual representation of a Sequence Diagram is formally defined as below.

**Definition 1** A Sequence Diagram is given by a three tuple \( \langle L, MSG, FG \rangle \), in which \( L \) is a non-empty set of Lifelines enclosed in the Sequence Diagram. \( MSG \) is a set of Messages directly enclosed in the Sequence Diagram, *i.e.*, Messages that are not contained by any CF. \( FG \) is a set of CFs directly enclosed in the Sequence Diagram, *i.e.*, the top level CFs, denoted as \( CF_1, CF_2, ..., CF_m \).

Messages that are directly enclosed in the top-level CFs will be defined in their respective CFs. Similarly CFs that are directly enclosed in top-level CFs are defined in their enclosing CFs. In this manner, a Sequence Diagram with CFs can be recursively defined.

**Definition 2** A Message has the form \( \langle name, mform, OS_s, OS_e \rangle \), where *name* is the Message name, *mform* denotes it is either a synchronous or an asynchronous Message, *OS_s* denotes its sending OS and *OS_e* denotes its receiving OS. Each OS has the form \( \langle loc, type \rangle \), where \( loc \) denotes its associated Lifeline, \( type \) denotes it is either a sending or a receiving OS.

Each Lifeline \( l \in L \) has a set of finite locations \( LOC(l) \subseteq \mathbb{N} \) on it. The locations form a finite sequence \( l_1, l_2, l_3, ..., l_k \), where \( k \in \mathbb{N} \). Each location is associated with an OS uniquely and vice versa, *i.e.*, the relation between set \( LOC(l_i) \) and the set returned by function \( OSS(l_i) \) is a one-to-one correspondence. Function \( OSS(l_i) \) returns the set of OSs on lifeline \( l_i \). For example, in figure 1b, the set \( LOC(l_2) \) contains seven locations, each of which is associated with an OS, *i.e.*, OSs \( r_1, s_2, r_3, s_4, r_5, r_6, r_7 \). Message \( msg \) is expressed by \( \langle m1, async, s_1, r_1 \rangle \), and OS \( s_1 \) is expressed by \( \langle l_1, 1, send \rangle \), where \( l_1 \) represents a participating object of class \( L_1 \).
**Definition 3** A CF $CF_m$ has the form $\langle L, \text{oper}, OP \rangle$. $L$ denotes the set of Lifelines enclosed by $CF_m$, including the Lifelines which may not intersect with the Messages of $CF_m$. \text{oper} denotes the Interaction Operator of $CF_m$. $OP$ denotes the sequence of Interaction Operands within $CF_m$, i.e., $\text{oper}_{m_1}, \text{oper}_{m_2}, \ldots, \text{oper}_{m_n}$. Each $\text{oper}_n \in OP$ has the form $\langle L, \text{MSG}, \text{FG}, \text{cond} \rangle$, where $L$ denotes the set of Lifelines enclosed by $\text{oper}_n$; $\text{MSG}$ denotes the set of Messages directly enclosed in $\text{oper}_n$; $\text{FG}$ denotes the set of CFs directly enclosed in $\text{oper}_n$; and $\text{cond}$ denotes the Interaction Constraint of $\text{oper}_n$, which is True if there is no Interaction Constraint. Without loss of generality, $\text{cond}$ is represented by a boolean variable. Comparing the structure between a Sequence Diagram and an Operand, the Sequence Diagram does not have an Interaction Constraint. In order for an Operand and a Sequence Diagram to share the same form, we assign an Interaction Constraint (which evaluates to True) to a Sequence Diagram.

Consider figure 1b as an example. Sequence Diagram $seq$ is represented by $\langle \{l_1, l_2, l_3\}, \{\text{msg}_1, \text{msg}_2\}, \{\text{CF}_1\} \rangle$, where the set of Lifelines enclosed by $seq$ contains three Lifelines, $l_1, l_2, l_3$, the set of Messages directly enclosed in $seq$ contains two Messages, $\text{msg}_1, \text{msg}_2$, and the set of CFs directly enclosed in $seq$ contains one CF, $\text{CF}_1$. $\text{msg}_1, \text{CF}_1$, and $\text{msg}_2$ are combined using Weak Sequencing. $\text{CF}_1$ is represented by $\langle \{l_1, l_2, l_3\}, \text{par}, \langle \text{op}_1, \text{op}_2 \rangle \rangle$, where $l_1, l_2, l_3$ are Lifelines enclosed by $\text{CF}_1$, $\text{par}$ is the Interaction Operator of $\text{CF}_1$, and $\text{op}_1, \text{op}_2$ are the Interaction Operands of $\text{CF}_1$. $\text{op}_1$ and $\text{op}_2$ preserve their execution order if their Interaction Constraints evaluate to True respectively, and the execution order between $\text{op}_1$ and $\text{op}_2$ are decided by Interaction Operator $\text{par}$. If both Constraints of $\text{op}_1$ and $\text{op}_2$ evaluate to False, $\text{CF}_1$ is excluded and Messages $\text{msg}_1$ and $\text{msg}_2$ are ordered by Weak Sequencing. Operand $\text{op}_1$ expresses the Messages and CFs directly enclosed in it, represented by $\langle \{l_1, l_2, l_3\}, \{\text{msg}_1\}, \{\text{CF}_2\}, \text{cond}_1 \rangle$, where $\text{cond}_1$ is $\text{op}_1$’s Interaction Constraint. In this way, the syntax of $seq$ is described recursively.

**B. Sequence Diagram Deconstruction**

To facilitate codifying the semantics of Sequence Diagrams and nested CFs in LTL formulas, we show how to deconstruct a Sequence Diagram and CFs to obtain fine-grained syntactic constructs. Eichner et al. have defined the Maximal Independent Set in [14] to deconstruct a Sequence Diagram into fragments, each of which covers multiple Lifelines. Their proposed semantics defines that entering a Combined Fragment has to be done synchronously by all the Lifelines, i.e., each Combined Fragment is connected with adjacent OSs and CFs using Strict Sequencing. Recall that CFs can be nested within other CFs. OSs and CFs directly enclosed in the same CF or Sequence Diagram are combined using Weak Sequencing, constraining their orders with respect to each individual Lifeline only [32]. To express the semantics of Weak Sequencing, we further deconstruct a Sequence Diagram into syntactic constructs on each Lifeline, which also helps us to define the semantics of nested CFs.

We project every CF $cf_m$ onto each of its covered Lifelines $l_i$ to obtain a compositional execution unit (CEU), which is denoted by $cf_m|l_i$. (The large dotted rectangle on Lifeline $L1$ in figure 2 shows an example).

**Definition 4** A CEU is given by a three tuple $\langle l_i, \text{oper}, \text{setEU} \rangle$, where $l_i$ is the Lifeline, onto which we project the CF, $\text{oper}$ is the Interaction Operator of the CF, and $\text{setEU}$ is the set of execution units, one for each Operand $\text{op}_n$ enclosed in the CF on Lifeline $l_i$.

Every Operand $\text{op}_n$ of CF $cf_m$ is projected onto each of its covered Lifelines $l_i$ to obtain an execution unit (EU) while projecting $cf_m$ onto $l_i$, denoted by $\text{op}_n|l_i$. If the projected Interaction Operand contains a nested Combined Fragment, a hierarchical execution unit (HEU) is obtained; otherwise a basic execution unit (BEU) is obtained, i.e., an EU is a BEU if it does not contain any other EUs. (The small dotted rectangle on Lifeline $L2$ in figure 2 shows an example of a BEU and the large dotted rectangle shows an example of an HEU).

**Definition 5** A BEU $u$ is given by a pair, $\langle \text{E}_u, \text{cond} \rangle$, in which $\text{E}_u$ is a finite set of OSs on Lifeline $l_i$ enclosed in Operand $\text{op}_n$, which are ordered by the locations associated with them, and $\text{cond}$ is the Interaction Constraint of the Operand. $\text{cond}$ is True when there is no Interaction Constraint.

**Definition 6** An HEU is given by $\langle \text{setCEU}, \text{setBEU}, \text{cond} \rangle$, where $\text{setCEU}$ is the set of CEUs directly enclosed in the HEU, i.e., the CEUs nested within any element of $\text{setCEU}$ are not considered. $\text{setBEU}$ is the set of BEUs that are directly enclosed in the HEU.

Projecting a Sequence Diagram onto each enclosing Lifeline also obtains an EU whose Constraint is True. The EU is an HEU if the Sequence Diagram contains CFs, otherwise, it is a BEU. In an HEU, we also group the OSs between two adjacent CEUs or prior to the first CEU or after the last CEU on the same level into BEUs, which inherit the parent HEU’s Constraint, $\text{cond}$. (The dotted rectangle on Lifeline $L1$ in figure 1b shows an example). The constituent BEU(s) and CEU(s) within an HEU execute sequentially, complying with their graphical order, as do the OSs in the BEU.
In the example of figure 1b, Lifeline L2 demonstrates the projections of the two CFs. The Parallel is projected to obtain a CEU. The first Operand of the Parallel is projected to obtain an HEU, containing the CEU projected from the Critical Region and the BEU composed of the sending OS of m2. The second Operand of the Parallel is projected to obtain a BEU. The CEU of the Critical Region contains a BEU projected from its single Operand. The OS prior to the Parallel is grouped into a BEU.

We provide a metamodel to show the abstract syntax of relations among BEUs, HEUs, and CEUs in figure 3. An EU can be a BEU or an HEU, and one or more EUs compose a CEU. An HEU contains one or more CEUs.

![Fig. 3. Execution Unit Metamodel](image)

C. Nested Combined Fragment

The syntactical definitions and deconstruction enable us to express the semantics of Sequence Diagram as a composition of nested CFs at different levels. We consider the OSs and CFs directly enclosed in the Sequence Diagram as the highest-level Interaction Fragments, which are combined using Weak Sequencing. These OSs are grouped into BEUs on each enclosing Lifeline, which observe total order within each BEU. For each Message, its sending OS must occur before its receiving OS. To enforce the interleaving semantics among Linelines, at most one OS may execute at a time within the Sequence Diagram. The semantics of the CFs are represented at a lower-level. Each CF contains one or more Operands, which are composed using the CF’s Interaction Operator. Each Interaction Operator determines its means of combining Operands without altering the semantics of each Operand. The semantics of an Operand within each CF are described at the next level. A Sequence Diagram can be considered as an Operand whose Constraint evaluates to True. Therefore, the semantics of each Operand containing other CFs can be described in the same way with that of a Sequence Diagram with nested CFs. An Operand containing no other CF is considered as the bottom-level, which has a BEU on each enclosing Lifeline. The Operand whose Constraint evaluates to False is excluded. In this way, the semantics of a Sequence Diagram with CFs can be described recursively.

IV. Defining Trace Semantics in LTL

The semantics of a Sequence Diagram is given by valid and invalid traces. Each trace is a sequence of OSs (i.e., event occurrences within the context of the Sequence Diagram). A Sequence Diagram model specifies complete traces, each of which describes a possible execution of the system, whereas a CF of the Sequence Diagram defines a collection of their subtraces. These subtraces may interleave with other OSs appearing in the Sequence Diagram but outside the CF, connecting using Weak Sequencing to make complete traces of the Sequence Diagram [35]. A trace derived from a Sequence Diagram can be finite, denoted as $\sigma[1..n] = \sigma_1\sigma_2...\sigma_n$. The trace derived from a Sequence Diagram can also be infinite if it expresses the behavior of infinite iterations in terms of Loop with infinity upper bound, denoted as $\sigma = \sigma_1\sigma_2...\sigma_n$.

This paper presents a framework to characterize the traces of Sequence Diagram in Linear Temporal Logic (LTL). LTL is a formal language for specifying the orders of events and states in terms of temporal operators and logical connectives. We use LTL formulas to express the semantic rules prescribed by Sequence Diagram constructs, each of which defines the execution orders among OSs. Note that an LTL formula represents infinite traces. In the case that a Sequence Diagram expresses a set of finite traces, we need to handle the mismatch between an LTL formula and a Sequence Diagram’s finite trace semantics. To bridge the gap, we adapt the finite traces of Sequence Diagrams without altering their semantics by adding stuttering of a no-op after the last OS $\sigma_n$ of each trace [17]. Then, LTL formulas can express these traces, each of which is denoted as $\Sigma^*_\text{seq} \tau^\omega$, where $\Sigma_{\text{seq}}$ is the set of OSs of seq and $\Sigma^*_\text{seq} \tau^\omega$ represents a finite sequence of OSs of seq. $\tau$ is a no-op, i.e., $\tau$ is an empty event occurrence and not observable, and $\tau^\omega$ represents an infinite sequence of no-ops.

A Sequence Diagram with Negative or Assertion CFs can specify desired properties as well as possible system executions in terms of traces. The Sequence Diagram for specifying desired properties only considers the OSs related to the properties. We represent the traces of properties with partial traces semantics, which allows other OSs do not appear in the Sequence Diagram but appear in the system executions to interleave the partial traces. Our framework supports partial traces semantics to express certain properties, including safety and consistency, with a Sequence Diagram.

We include a summary of temporal operators that are sufficient to understand our LTL template. $\Box p$ means that formula $p$ will continuously hold in all future states. $\diamond p$ means that formula $p$ holds in some future state. $\Diamond p$ means formula $p$ holds in the next state. $\Diamond p$ means that formula $p$ holds in the previous state. $\triangle p$ means that formula $p$ holds in some past state. $\triangledown p$ means that formula $p$ holds in some past state, excluding current state. $p \cup q$ means that formula $p$ holds until some future state where $q$ becomes true, and $p$ can be either True or False at that state. The macro $p \cup q \equiv p \cup (q \land p)$ states that in the state when $q$ becomes True, $p$ stays True.

V. Specifying Sequence Diagram in LTL

In this section, we describe how to use LTL formulas to codify the semantic rules of Sequence Diagrams as shown in section II. Formalizing the semantics of a notation can be challenging, especially if we consider all semantic constraints.
at once. To reduce the complexity and to improve the readability, we devise an LTL framework, comprised of simpler definitions, which we call templates, to represent each semantic aspect (i.e., the execution order of event occurrences imposed by individual constructs) as a separate concern. To capture the meanings of nested CFs, we provide a recursively defined template, in which each individual CF’s semantics is preserved (e.g., the inner CF’s semantics is not altered by other CFs containing it). These templates can then be composed using temporal logic operators and logical connectives to form a complete specification of a Sequence Diagram. In this way, if the notation evolves, many of the changes can still be localized to respective LTL templates.

To facilitate the representation of a Sequence Diagram in LTL, we define a collection of auxiliary functions (see table I) to access information of a Sequence Diagram. We provide the algorithms to calculate some auxiliary functions in Appendix A. These functions are grouped into two categories. The functions within the first group return the syntactical constructs of a Sequence Diagram. For instance, function $SN(j)$ returns the sending OS of Message $j$. The functions within the second group return the constructs, either whose Constraints evaluate to True or which are contained in the constructs whose Constraints evaluate to True. For instance, for Parallel CF1 in figure 1b, function $nested(CF1)$ returns a singleton set containing Critical Region $CF2$ if the Constraint of the first Operand of $CF1$ evaluates to True. Otherwise, $nested(CF1)$ returns an empty set, and Critical Region $CF2$ is ignored to reflect the semantic rule 3 which is general to all CFs (see section II-B). Functions $MSG(p)$, $LN(p)$, $AOS(q)$ are overloaded where $p$ can be an Interaction Operand, a CF, or a Sequence Diagram, and $q$ can be $p$, an EU, or a CEU.

### A. Basic Sequence Diagram

We start with defining an LTL template, called $\Pi_{seq}^{Basic}$ (see figure 4), to represent the semantics of a basic Sequence Diagram. The semantic rules for basic Sequence Diagram $seq$ defined in section II-A are codified separately using formulas $\alpha_g$, $\beta_j$, and $\varepsilon_{seq}$.

$\alpha_g$ focuses on the intra-lifeline behavior to enforce rules 1 and 2. Recall that when projecting basic Sequence Diagram $seq$ onto its covered Lifelines, $LN(seq)$, we obtain BEU $g$ for each Lifeline $i$, denoted as $seq\upharpoonright_i$. Each BEU $g$ contains a trace of OSs, $\sigma\upharpoonright_{(r+|AOS(g)|-1)}$, where $(r \geq 0)$ and $\sigma_r$ is the first OS in BEU $g$, function $AOS(g)$ returns the set of OSs within $g$, and $|AOS(g)|$ has its usual meaning, returning the size of set $AOS(g)$. The first conjunct of $\alpha_g$ enforces the total order of OSs in each BEU $g$, i.e., for all $k \geq r$, OS$_k$ must happen (strictly) before OS$_{k+1}$, ensured by $\neg OS_{k+1} \cup OS_k$. The second conjunct of $\alpha_g$ enforces that every OS in BEU $g$ executes only once. The semantics enforced by each $\alpha_g$ does not constrain each other. Thus, the intra-lifeline semantics of $seq$ is enforced by the conjunction of $\alpha_g$ for each Lifeline. Similarly, the semantics rule 3 is codified by a conjunction of $\beta_j$ for each Message $j$. Formula $\beta_j$ enforces that, for Message $j$, its receiving OS, $RCV(j)$, cannot happen until its sending OS, $SND(j)$, happens. Formula $\varepsilon_{seq}$ enforces interleaving semantics of complete traces among all the OSs of Sequence Diagram $seq$ in the fourth rule, which denotes that only one OS of $seq$ can execute at once, and the trace should execute uninterrupted until all the OSs of $seq$ have taken place. The trace stutters at the end with $\tau$. We define the logical operator “unique or” as “\(\lor\)”, to denote that exactly one of its OSs is chosen. A formula with logical connectives, $\bigwedge_{a_i \in S} a_i$, returns the conjunction of all the elements $a_i$ within the set $S$. It returns True if $S$ is an empty set.

### B. Combined Fragments

A Combined Fragment (CF) can modify the sequential execution of its enclosed OSs on each Lifeline. Moreover, a Sequence Diagram can contain multiple CFs that can be nested within each other. Though these features make a Sequence Diagram more expressive, they increase the complexity of representing all the traces of CFs. To capture these features, we generalize $\Pi_{seq}^{Basic}$ to $\Pi_{seq}$ for expressing Sequence Diagram with CFs (see figure 5). We introduce a new template $\Phi^{CF}$ to assert the semantics of each CF directly enclosed in $seq$.  

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LN(p)$</td>
<td>return the set of all Lifelines in $p$.</td>
</tr>
<tr>
<td>$MSG(p)$</td>
<td>return the set of all Messages directly enclosed in $p$.</td>
</tr>
<tr>
<td>$SND(j)$</td>
<td>return the sending OS of Message $j$.</td>
</tr>
<tr>
<td>$RCV(j)$</td>
<td>return the receiving OS of Message $j$.</td>
</tr>
<tr>
<td>$Reply(u)$</td>
<td>return the reply Message of a synchronous Message containing OS $u$.</td>
</tr>
<tr>
<td>$typeOS(u)$</td>
<td>return the type of OS $u$, which is a sending OS or a receiving OS.</td>
</tr>
<tr>
<td>$typeCF(u)$</td>
<td>return the Interaction Operator of CF $u$.</td>
</tr>
<tr>
<td>$TOP(u)$</td>
<td>return the set of Interaction Operands whose Constraints evaluate to True within CF $u$, i.e., ${op</td>
</tr>
<tr>
<td>$nested(u)$</td>
<td>return the set of CFs, which are directly enclosed in CF $u$’s Interaction Operands whose Constraints evaluate to True. It can be overloaded to an Interaction Operand or a Sequence Diagram.</td>
</tr>
<tr>
<td>$TBEU(u)$</td>
<td>for CEU or EU $u$, return a set of directly enclosed BEUs, whose Constraints evaluate to True, i.e., ${beu</td>
</tr>
<tr>
<td>$AOS(q)$</td>
<td>return the set of OSs which are enabled (i.e., the Constraints associated with it evaluate to True) and chosen to execute in $q$.</td>
</tr>
<tr>
<td>$TOS(u)$</td>
<td>return the set of OSs of the BEUs directly enclosed in CEU or EU $u$ whose Constraints evaluates to True, i.e., ${os</td>
</tr>
<tr>
<td>$pre(u)$</td>
<td>return the set of OSs which may happen right before CEU $u$. The set contains an OS if a BEU whose Constraints evaluates to True prior to $u$ on the same Lifeline. If a CEU executes prior to $u$ on the same Lifeline, the set may contain a single or multiple OSs depending on the CEU’s Operator and nested CEUs (if there are any nested CEUs). If an HEU executes prior to $u$ on the same Lifeline, the set is determined by the last CEU or BEU nested within the HEU.</td>
</tr>
<tr>
<td>$post(u)$</td>
<td>return the set of OSs which may happen right after CEU $u$, which can be calculated in a similar way as $pre(u)$.</td>
</tr>
</tbody>
</table>
Template $\Pi_{\text{seq}}$ is a conjunction of the formulas $\alpha_g$, $\beta_j$, $\Phi^{CF}$ and $\varepsilon_{\text{seq}}$, which is equivalent to the LTL template of basic Sequence Diagram if seq does not contain any CF.

When multiple CFs and OSs present in a Sequence Diagram, they are combined using Weak Sequencing — CFs and OSs on the same Lifeline execute sequentially, whereas CFs and OSs on different Lifelines execute independently, except the pairs of OSs belonging to Messages. Thus, we project Sequence Diagram seq with CFs onto Lifelines to obtain a collection of CEUs and EUs, facilitating us to focus on OSs on each single Lifeline. The OSs directly enclosed in seq are grouped into BEUs, whose semantics are enforced by a conjunction of $\alpha_g$ for each BEU g. The order of OSs within Messages directly enclosed in seq are enforced by a conjunction of $\beta_j$ for each Message j. $\varepsilon_{\text{seq}}$ enforces that at most one OS can execute at a time for all the OSs within seq. One way to implement these formulas is provided in Appendix B. If seq contains a Loop, the OSs of seq includes OSs in each iteration of the Loop.

Template $\Phi^{CF}$ (see figure 6) considers three cases. Formula (1) asserts the case that the CF contains no Operand whose Constraint evaluates to True. Thus, the OSs within the CF are excluded from the traces. Semantics rule 3 for CFs states Weak Sequencing among the CF’s preceding Interaction Fragments and succeeding ones, which is enforce by formula $\eta^{CF}$. Functions pre(CF $\uparrow_i$) and post(CF $\uparrow_i$) return the set of OSs which may happen right before and after CEU CF $\uparrow_i$ respectively. The formula $\eta^{CF}$ enforces that the preceding set of OSs must happen before the succeeding set of OS on each Lifeline i, which sets to True if either pre(CF $\uparrow_i$) or post(CF $\uparrow_i$) returning empty set. Formula (2) asserts the case that CF contains at least one Operand whose Constraint evaluates to True, and CF is not an Alternatives or a Loop. The first conjunct $\Psi^{CF}$ defines the semantics of OSs directly enclosed in CF. The second conjunct states that the semantics of each CFi, which is directly enclosed in CF, is enforced by each $\Phi^{CFi}$. In this way, $\Phi^{CF}$ can be defined recursively until it has no nested CFs.

Template $\Psi^{CF}$ captures the semantics that is common to all CFs (except Alternatives and Loop) (see figure 7). Sub-formula $\gamma^C_Fi$ enforces semantic rule 1, which defines the sequential execution on every Lifeline i. The first conjunct enforces that the preceding set of OSs must happen before each OS in CF on Lifeline i, and the second conjunct enforces that the succeeding set of OSs must take place afterwards. $\theta^{CF}$ states semantic rule 2, which defines the order among OSs directly enclosed in CF. $\theta^{CF}$ is a conjunction of $\alpha_g$s and $\beta_j$s. The $\alpha_g$s is a conjunction of all $\alpha_g$ of each Lifeline, where g is a BEU whose Constraint evaluates to True. The $\beta_j$s is a conjunction of $\beta_j$ of each Message.

Formula (3) asserts the case for Alternatives and Loop, which contain at least one Operand whose Constraint evaluates to True. For Alternatives, $\Psi^{CF}_{alt}$ defines the semantics of OSs and CFs directly enclosed in CF. $\Psi^{CF}_{alt}$ and $\Phi^{CF}$ for CFi directly nested in the Alternatives form an indirect recursion (see figure 10). The semantics of Loop is defined in a similar way (see figure 15).

The semantic rule varies for CFs with different Operators, which is enforced by adding different semantics constraints on $\Psi^{CF}$ for each individual CF respectively. The semantics specifics for different types of CF Operators are defined as below.

1) Concurrency: The Parallel represents concurrency among its Operands. The OSs of different Operands within Parallel can be interleaved as long as the ordering imposed by each Operand is preserved. Figure 1b is an example of Parallel with two Operands. The OSs within the sameOperand respect the order along a Lifeline or a Message, whereas the OSs from different Operands may execute in any order even if they are on the same Lifeline. For instance, OS r5 (i.e., the receiving OS of Message m5) and OS r6 on Lifeline L2 maintain their order. OS r2 and OS s5 on Lifeline L1 many execute in any order since they are in different Operands. Parallel does not add extra constraint to the general semantic rules of CF. Thus, the semantics of Parallel can be formally defined,

$$\Psi^{CF}_\text{par} = \theta^{CF} \land \bigwedge_{i \in LN(CF)} \gamma^C_Fi$$

2) Branching: Collectively, we call Option, Alternatives, and Break branching constructs.

a) Representing Option: The Option represents a choice of behaviors that either the (sole) Operand happens or nothing happens. As Option does not add any extra constraint to the execution of its sole Operand, its semantics can be formally defined as the template,
The Weak Sequencing of the Alternatives is recursion. The sub-formula of the unchosen Operand (else Functions semantics of CFs directly enclosed in the chosen Operand.

The translation of an Alternatives into an LTL formula must Constraints evaluate to True, the unchosen Operand does not add
exclusively or an implicit or an “else” Constraint. The chosen
Operand’s Constraint must evaluate to True. An implicit
Constraint always evaluates to True. The “else” Constraint is
the negation of the disjunction of all other Constraints in the
enclosing Alternatives. If none of the Operands whose
Constraints evaluate to True, the Alternatives is excluded.
The translation of an Alternatives into an LTL formula must
enumerate all possible choices of executions in that only
OSs of one of the Operands, whose Constraints evaluate to
True, will happen. LTL formula \( \Phi_{\text{alt}} \) in Figure 10 defines the
semantics of Alternatives, which is a conjunction of \( \Phi_{\text{alt}} \). Each
\( \Phi_{\text{alt}} \) represents the semantics of Operand \( m \), whose Constraint
evaluates to True, which is achieved by function \( \text{TOP}(\text{CF}) \).

The semantics of the chosen Operand (if clause) is described by \( \Phi_{\text{alt}} \) and \( \Phi_{\text{CF}} \), where \( \Phi_{\text{alt}} \) defines the partial order of
OSs within the chosen Operand and \( \Phi_{\text{CF}} \) defines the
semantics of CFs directly enclosed in the chosen Operand.
Functions \( \Phi_{\text{alt}} \) and \( \Phi_{\text{CF}} \) invoke each other to form indirect recursion. The sub-formula of the unchosen Operand (else clause) returns True, i.e., the unchosen Operand does not add
any constraint. The Weak Sequencing of the Alternatives is
represented by \( \gamma_{\text{alt}}^{\text{CF}} \) instead of \( \gamma_{\text{alt}}^{\text{CF}} \), which enforces Weak
Sequencing between the chosen Operand and the preceding/succeeding OSs of the Alternatives.

One way to implement the chosen Operand (if clause) is using a boolean variable \( \text{exe} \) for each Operand whose Interaction Constraint evaluates to True. The variable \( \text{exe} \) should satisfy the following assertion,

\[
\bigwedge_{i \in [1..m]} \text{exe}_i \land \bigwedge_{i \in [1..m]} (\text{exe}_i \rightarrow \text{cond}_i)
\]

The first conjunct expresses that only one \( \text{exe} \) sets to True, i.e., exactly one Operand is chosen. The second conjunct enforces that the Interaction Constraint of Operand whose \( \text{exe} \) sets to True must evaluate to True. Figure 9 shows an example of an Alternatives with three Operands enclosing three Lifelines. We assume the Constraints of the first and the third Operands evaluate to True, the one of the second Operand evaluates to False. Only one between the first and the third Operands is chosen by evaluating its variable \( \text{exe} \) to True.

\[
\Phi_{\text{CF}} = \theta_{\text{CF}} \land \bigwedge_{i \in \text{LN}(\text{CF})} \gamma_{\text{i}}^{\text{CF}}
\]

\[
\theta_{\text{CF}} = \bigwedge_{i \in \text{LN}(\text{CF})} \left( \bigwedge_{j \in \text{MSG}(\text{TOP}(\text{CF}))} \alpha_j \right) \land \bigwedge_{j \in \text{MSG}(\text{TOP}(\text{CF}))} \beta_j
\]

\[
\gamma_{\text{i}}^{\text{CF}} = \bigwedge_{\text{OS} \in \text{TOS}(\text{CF}^{\text{i}})} \left( \bigwedge_{\text{OS}_{\text{pre}} \in \text{pre}(\text{CF}^{\text{i}})} \left( -\text{OS}_{\text{post}} \right) \right) \bigwedge_{\text{OS}_{\text{pre}} \in \text{pre}(\text{CF}^{\text{i}})} \left( \bigwedge_{\text{OS}_{\text{post}} \in \text{post}(\text{CF}^{\text{i}})} \left( \text{OS}_{\text{pre}} \right) \right)
\]
c) **Representing Break:** The Break states that if the Operand's Constraint evaluates to *True*, it executes instead of the remainder of the enclosing Interaction Fragment. Otherwise, the Operand does not execute, and the remainder of the enclosing Interaction Fragment executes. A Break can be represented as an Alternatives in a straightforward way. We rewrite the semantics interpretation of Break as an Alternatives with two Operands, the Operand of Break and the Operand representing the remainder of the enclosing Interaction Fragment. The Constraint of the second Operand is the negation of the first Operand’s Constraint. For example, the Interaction Fragment enclosing the Break is the first Operand of the Parallel rather than the Parallel (see figure 11). We rewrite the Sequence Diagram, using Alternatives to replace Break (see figure 12). cond3 is the Constraint of Break and cond4 is the negation of it. In this way, only one Operand can be chosen to execute. Thus, the LTL representation of Break can be represented as the LTL formula for Alternatives with two Operands.

3) **Atomicity:** The Critical Region represents that the execution of its OSs is in an atomic manner, i.e., restricting OSs within its sole Operand from being interleaved with other OSs on the same Lifeline. In the example of figure 1b, a Critical Region is nested in the first Operand of the Parallel. OSs s2, r5 and r6 can not interleave the execution of OSs r3 and s4. Formula $\Psi_{\text{critical}}^{CF}$ presents the semantics for Critical Region (see figure 13). $\theta_{CF}$ and $\gamma_{i}^{CF}$ have their usual meanings. $\delta_{M_{1},M_{2}}$ enforces that on each Lifeline, if any of the OSs within the CEU of Critical Region (representing as the set of $M_{1}$) occurs, no other OSs on that Lifeline (representing as the set of $M_{2}$) are allowed to occur until all the OSs in $M_{1}$ finish. Thus, $M_{1}$ is guaranteed to execute as an atomic region. Function “/” represents the removal of the set of OSs for Critical Region from the set of OSs for Sequence Diagram seq on Lifeline i.

4) **Iteration:** The Loop represents the iterations of the sole Operand, which are connected by Weak Sequencing. To restrict the number of iterations, the Operand’s Constraint may include a lower bound, minint, and an upper bound, maxint, i.e., 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 minint number of iterations, the Loop will terminate. First, we consider fixed Loop. Figure 14 is an example of fixed Loop which iterates exactly three times.

Each OS is an instance of an event, which is unique within a Sequence Diagram. To keep each OS within different iterations of a Loop unique, one way to implement an OS is defining an array to rename the OS of each iteration. We introduce $R$, representing the number of iterations and $n$, representing the current iteration number on Lifeline i. The Loop in iteration $n$ can be represented as $\text{Loop}[n]$. For example, the Loop in figure 14 has three iterations, $\text{Loop}[1]$, $\text{Loop}[2]$ and $\text{Loop}[3]$. Figure 15 shows an LTL formula for a Loop. $\theta_{R}$ overloads $\theta_{CF}$, which asserts the order of OSs during each iteration. $\gamma_{i,R}$ enforces the Weak Sequencing among Loop iterations and its preceding/following sets of OSs on each Lifeline $i$, i.e., the first Loop iteration execute before the preceding set of OSs, and the last Loop iteration execute after the succeeding set of OSs. An OS and the value of $n$ together represent the OS in a specific iteration, (e.g., the element $(OS_{k}[n])$ expresses $OS_{k}$ in the $n$th iteration). The OSs within nested CFs are renamed with the same strategy. Template $\kappa_{i,R}$ is introduced to enforce Weak Sequencing among Loop iterations, e.g., on
the same Lifeline, $OS_j[n + 1]$ can not happen until $OS_k[n]$ finishes execution.

To represent the semantics of Loop with infinity upper bound, we generalize the propositional LTL formulas to first-order LTL formulas to express that the number of iterations is infinite. Figure 16 shows the LTL formula of Loop with infinity upper bound. For infinite iterations, $\theta_{infty}$ asserts the order of OSs during each iteration, and $\kappa_{i,infty}$ enforces Weak Sequencing among the iterations. $\gamma_{i,infty}$ enforces the Weak Sequencing between the Loop’s preceding set of OSs and the first Loop iteration on each Lifeline $i$. (The Loop with infinity upper bound has no succeeding set of OSs.)

If the Loop is not fixed and it does not have infinity upper bound, we need to evaluate the Interaction Constraint of the its sole Operand during each iteration. Similarly to fixed Loop, the finite but not fixed Loop can be unfolded by repeating iterations. To keep the Constraint of each iteration unique, an array is defined to rename the Constraint, e.g., the Constraint of iteration $n$ is represented as $cond[n]$. The order of OSs during each iteration is asserted as the fixed Loop. Two adjacent iterations are connected using Weak Sequencing. If $n \leq \minint$, $cond[n]$ sets to $True$ and the Loop executes. If $\minint < n \leq \maxint$, the Loop executes only if $cond[n]$ evaluates to $True$. Otherwise, the Loop terminates and the Constraints of remaining iterations (i.e., from $cond[n + 1]$ to $cond[\maxint]$) set to $False$. The Loop no longer executes when its iteration reaches $\maxint$.

5) Negation: A Negative represents that the set of traces within a Negative are invalid. For example, there are three traces defined by the Negative in figure 17 $[s1, s2, r1, r2]$, $[s2, s1, r1, r2]$, and $[s1, r1, s2, r2]$, which are invalid traces. Formula $\Psi_{\neg CF} = \theta_{CF}$ formally defines the semantics of Negative CF, asserting the order of OSs directly enclosed in it. If the Interaction Constraint of the Negative evaluates to $False$, the traces within the Negative may be either invalid traces or the Operand is excluded (see subsection V-D1 for details).

6) Assertion: An Assertion representing, on each Lifeline, a set of mandatory traces, which are the only valid traces following the Assertion’s preceding OSs. Its semantics is formally defined as $\Psi_{assert}$ in figure 19. $\theta_{CF}$ and $\gamma_{i}$ have their usual meanings. Function $\lambda_{i,seq}^{pre,\ell,\gamma_{i}}(AOS(CF_{i,1}, AOS(seq_{i,1}) \setminus AOS(CF_{i,1})))$ represents that on Lifeline $i$, if all the OSs in the set of $pre$ happen, no other OSs in Sequence Diagram $seq$ are allowed to happen until all the OSs in assertion complete their execution. The function prevents the Assertion and its preceding OSs from being interleaved by other OSs, which is required when the Assertion is nested within other CFs, such as Parallel. For example (see figure 18), an Assertion is nested within a Parallel. The OSs within the CEU of the Assertion execute
The Weak Sequencing restricts the execution orders among its Operands along each Lifeline. Figure 20 is an example of Weak Sequencing, where OS s4 can not happen until OS s3 execute, whereas OS s4 and r3 may happen in any order as they are on different Lifelines. The LTL definition of Weak Sequencing is given as below.

\[ \psi_{weak}^{CF} = \theta_{CF} \land \bigwedge_{i \in LN(CF)} \gamma_{(i+1),CF} \land \bigwedge_{i \in LN(CF)} \gamma_{(i+1),CF} \land \bigwedge_{m \in TOP(CF)} \gamma_{m}^{n} \]

 Templates \( \theta_{CF} \) and \( \gamma_{i}^{CF} \) have their usual meanings. \( \gamma_{i}^{n} \) specifies the execution orders between adjacent Operands, as well as enforcing the Weak Sequencing between the CF and its preceding/succeeding Interaction Fragments \( \gamma_{i}^{CF} \). (The LTL formula keeps \( \gamma_{i}^{CF} \) for clarity and consistency.)

8) Strict Sequencing: The Strict Sequencing imposes an order among OSs within different Operands. For an Operand, all OSs must take place before any OS of its following Operand. In other words, any OS of an Operand can not execute until all OSs of the previous Operand finish execution. The Strict Sequencing enforces the synchronization among multiple Lifelines, i.e., any covered Lifeline needs to wait other Lifelines to enter the second or subsequent Operand to enter the second or subsequent Operand to complete execution.

Figure 22 presents the semantics of Strict Sequencing. Template \( \theta_{CF}^{∗} \) has its usual meaning. The Strict Sequencing
9) Coregion: A Coregion is an area of a single Lifeline, which is semantically equivalent to a Parallel that the OSs are unordered. Figure 23 shows an example of Coregion, where OS r3 and r4 may execute in any order. We represent the Coregion into an LTL formula in a similar way with a Parallel. Each OS within the Coregion is considered as an Operand of Coregion whose Constraints evaluate to True. Figures 21, 22, and 23 each show the ordered and unordered Coregions.

\[ \psi^{CP}_{\text{coregion}} = \gamma^{CP}_i \]

C. Ignore and Consider

So far, all the CFs define a collection of partial traces, which only interleave the OSs appearing in the Sequence Diagram to form a complete trace. The Ignore and Consider CFs allow other OSs that are not considered or ignored extend the traces. Ignore and Consider take into consideration the message types which do not appear in the Sequence Diagram. Generally, the interpretation of a Sequence Diagram only considers the message types explicitly shown in it. An Ignore specifies a list of message types which needs to be ignored within the CF. For instance, Messages whose type is m3 are ignored in the Ignore CF (see figure 24). A Consider specifies a list of considered message types, which is equivalent to specifying other possible message types to be ignored. For instance, the Consider CF only considers Messages whose types are m2, m3 or m5 (see figure 25). To design well-formed Ignore or Consider, some syntactical constraints need to be mentioned. For Consider, only Messages whose types specified by the list of considered Messages can appear in the CF [35]. For Ignore, the ignored message types are suppressed in the CF [35].

Within the Ignore, the Messages appearing in the CF and the Messages which are explicitly ignored in the CF need to be constrained (see figure 26). \( \theta^{CP} \) and \( \gamma^{CP}_i \) have their usual meanings, which describe the semantics of Messages appearing in the Ignore. Each OS of the ignored Messages
executes only once, which is enforced by $\bar{\alpha}_{\text{ignoreOS(CF)}}$. We introduce function $\text{ignoreMsg}(\text{CF})$ to return the set of Messages of the ignored message types which occur in $\text{CF}$, which can be finite or infinite. Function $\text{ignoreOS}(\text{CF})$ returns the set of OSs associated with Messages of ignored message types, which can also be finite or infinite. Formula $\beta_k$ enforces that, for each ignored Message $k$, its sending OS must happen before its receiving OS. Formula $\gamma_i^{\text{CF}, \text{ignoreOS}(\text{CF}1)}$, which enforces any OS of the set of the ignored OSs can only happen within the CEU of the Ignore on each Lifeline, formally,

$$
\gamma_i^{\text{CF}, \text{ignoreOS}(\text{CF}1)} = \bigwedge_{\text{OS} \in \text{OS}(\text{S})} ((\neg \text{OS}) \bar{U} \bigwedge_{\text{OS}_{\text{pre}} \in \text{pre}(\text{CF}1)} (\Diamond \text{OS}_{\text{pre}})) \land ((\bigwedge_{\text{OS}_{\text{post}} \in \text{post}(\text{CF}1)} (\neg \text{OS}) \bar{U} (\Diamond \text{OS})))
$$

where $\text{S}$ can be replaced using $\text{ignoreOS}(\text{CF}1)$. Formula $\varepsilon_{\text{seq}, \text{ignoreOS}(\text{CF})}$ extends $\varepsilon_{\text{seq}}$ to include the OSs of ignored Messages in the set of OSs of $\text{seq}$, formally,

$$
\varepsilon_{\text{seq}, \text{ignoreOS}(\text{CF})} = \square((\bigwedge_{\text{OS}_{\text{p}} \in (\text{AOS}(\text{seq}) \cup \text{ignoreOS}(\text{CF}))} \text{OS}_{\text{p}}) \land (\bigwedge_{\text{OS}_{\text{p}} \in (\text{AOS}(\text{seq}) \cup \text{ignoreOS}(\text{CF}))} (\Diamond \text{OS}_{\text{p}})))
$$

Thus, function $\varepsilon_{\text{seq}}$ of Sequence Diagram with Ignore enforces the interleaving semantics among OSs appearing in $\text{seq}$ and OSs of the ignored Messages.

As the dual Operator of $\text{ignore}$, the semantics of a CF with Operator $\text{consider}$ is equivalent to ignoring all possible message types except the considered types. In this way, the LTL formula of Ignore can be adapted to represent the semantics of Consider (see figure 27). Function $\text{AllMsg}(\text{CF}) \setminus \text{considerMsg}(\text{CF})$ returns the Messages which are not considered but occur in $\text{CF}$, where $\text{AllMsg}(\text{CF})$ returns all possible Messages, including Messages of considered types and Messages of ignored types. $\text{considerMsg}(\text{CF})$ returns the Messages of considered types. Function $\Sigma \setminus \text{considerOS}(\text{CF})$ returns all possible OSs within $\text{CF}$ except the OSs of considered Messages, where $\Sigma$ is the set of all possible OSs including considered OSs and ignored OSs, and $\text{considerOS}(\text{CF})$ returns the set of OSs of considered Messages. In this way, the Sequence Diagram with Consider or Ignore no longer derive complete traces.

D. Semantic Variations

OMG provides the formal syntax and semi-formal semantics for UML Sequence Diagrams, leaving semantic variation points for representing different applications. Micskei and Waeselynck have collected and categorized the interpretations of the variants [28]. In the following subsections, we discuss how to use our LTL framework to formalize the variations of Negative, Strict Sequencing, and Interaction Constraints.

1) Variations of Negative: Recall that the traces defined by a Negative are considered as invalid traces. For example, if the Operand of Negative $S$, which does not contain any other Negative, defines a set of valid traces, then the set of traces defined by $S$ are invalid traces. In the case that the Constraint of the Operand of $S$ evaluates to $\text{False}$, the interpretation of the semantics of $S$ may be varied, depending on the requirement of applications. Formula $\Psi_{\text{neg}}^S$ instantiates the template $\Psi^S$ (see subsection V-B5) with $S$, defining the traces of $S$, which can be invalid or inconclusive. For example, three traces defined by the Negative (see figure 17), $[s1, s2, r1, r2]$, $[s2, s1, r1, r2]$, and $[s1, r1, s2, r2]$, can be interpreted as invalid, or inconclusive traces if $\text{cond}$ evaluates to $\text{False}$.

In the case that, Negative $S$ is enclosed in Sequence Diagram or non-Negative CF $R$, the Messages which are not enclosed in $S$ may interleave the sub-traces of $S$. If the sub-traces of $S$ are invalid, the traces of $R$ can be interpreted as invalid or inconclusive traces. If the sub-traces of $S$ are inconclusive traces (i.e., the Constraint of the Operand of $S$ evaluates to $\text{False}$), the traces of $R$ are also inconclusive traces. For Sequence Diagram $R$, its traces are defined by formula $\Pi_R$, which instantiates the template $\Pi_{\text{seq}}$ (see figure 5). For non-Negative CF $R$, its traces are defined by formula $\Phi^R$, which instantiates the template $\Phi^\text{CF}$ (see figure 6). For example, trace $[s1, s2, r2, r1, s3, r3]$ in figure 28 is interpreted as an invalid or an inconclusive trace.

For nested Negative CFs, i.e., Negative CF $R$ encloses Negative CF $S$, the traces of $R$ are defined by $\Phi^R$. These traces can be interpreted as valid, invalid, or inconclusive traces, depending on the Constraint of $R$’s Operand and the
interpretation of the sub-traces of $S$. The sub-traces of $S$ are invalid or inconclusive depending on the value of its Constraint. Three different interpretations for the traces of $R$ are provided: (1) If the sub-traces of $S$ are invalid traces and the Constraint of $R$’s Operand evaluates to $\text{True}$, the traces of $R$ can be valid, invalid, or inconclusive traces. (2) If the sub-traces of $S$ are invalid traces and the Constraint of $R$’s Operand evaluates to $\text{False}$, the traces of $R$ can be invalid or inconclusive traces. (3) If the sub-traces of $S$ are inconclusive, the traces of $R$ can be inconclusive traces in despite of the evaluation. Figure 29 shows an example of nested Negative CFs. All the traces $[s_1, s_2, r_1, r_2], [s_2, s_1, r_1, r_2]$, and $[s_1, r_1, s_2, r_2]$ of $R$ can be valid, invalid, or inconclusive traces depending on the value of $\text{cond1}$ and $\text{cond2}$.

2) Variations of Strict Sequencing: A Strict Sequencing CF represents an order among its Operands that any OS in an Operand can not execute until the previous Operand completes execution. However, the connection between the Strict Sequencing and its preceding/succeeding Interaction Fragments can be varied. According to the semantic rules general to all CFs, the Strict Sequencing is connected with its preceding/succeeding Interaction Fragments using Weak Sequencing. However, some applications may require that the Strict Sequencing are connected with its preceding/succeeding Interaction Fragments using Strict Sequencing. We modify the LTL formula of Strict Sequencing to formalize the variation (see figure 30). The only change we need to make is to replace $\gamma_i^{CF}$ that enforces Weak Sequencing between the Strict Sequencing and its preceding/succeeding Interaction Fragments with $\nu_i^{CF}$. Function $\nu_i^{CF}$ enforces the synchronization among multiple Lifelines when entering or leaving the Strict Sequencing, $\text{i.e.}$, any covered Lifeline needs to wait others to enter or leave the Strict Sequencing together. The first conjunct enforces that the preceding set of OSs must happen before each OS within the Strict Sequencing, and the second conjunct enforces that the succeeding set of OSs must take place afterwards.

If an application requires Strict Sequencing to connect any CF with its preceding/succeeding Interaction Fragments, we can use function $\nu_i^{CF}$ to replace function $\gamma_i^{CF}$ in the LTL formula of the CF.

3) Variations of Interaction Constraint: There are two semantic interpretations of an Operand whose Interaction Constraint evaluates to $\text{False}$: (1) The Operand is excluded and its traces are inconclusive; (2) The traces expressed by the Operand are interpreted as invalid traces. Our LTL template chooses the first interpretation since that is the semantics provided by OMG (page 488 in [32]). For the second interpretation, the semantics of the Operand whose Constraint evaluates to $\text{False}$ can be defined in the same way as the semantics of Negative CF (see subsection V-D1).

E. Other Control Constructs

1) General Ordering: General Ordering imposes order of two unordered OSs. We specify the two OSs of General Ordering as a pair of ordered OSs. In the LTL formula of General Ordering, $OS_p$ and $OS_q$ are two OSs connected by the General Ordering, which specifies that $OS_q$ can not execute until $OS_p$ completes execution.

$$\Upsilon^{GO} = \neg OS_q \ U OS_p$$

2) Interaction Use: Interaction Use embeds the content of the referred Interaction into the specified Interaction, thus composing a single, larger Interaction. We consider Interaction Use as a type of CF whose Interaction Operator is $\text{ref}$. Formula $\Psi_{ref}^{CF}$ represents the LTL representation of an Interaction Use. In $\Psi_{ref}^{CF}$, the first conjunct describes that the OSs directly enclosed in the referred Sequence Diagram obeys their order. The second conjunct enforces that the referred Sequence Diagram and its adjacent OSs are ordered by Weak Sequencing, which is represented by $\gamma_i^{CF}$.

$$\Psi_{ref}^{CF} = \theta^{CF} \land \bigwedge_{i \in LN(CF)} \gamma_i^{CF}$$
A nested CF.

the first one to execute only if it is located above an OS, not assume that an Interaction Constraint can restrict an OS to be Parallel, are located on and OS

Fig. 30. LTL formula for variation of Strict Sequencing

\[
\Psi_{\text{strict}}^{CF} = \theta_{CF}^{CF} \land \bigwedge_{k \in NFTOP(CF)} \chi^k \land \nu_{CF}^{CF} = \left( \left( \bigwedge_{i \in LN(CF)} \left( \bigwedge_{OS \in TOS(CF_1)} \neg OS \right) \right) \bigcup \left( \bigwedge_{i \in LN(CF)} \left( \bigwedge_{OS_{post} \in post(CF_1)} \neg OS_{post} \right) \right) \right) \bigcup \left( \bigwedge_{i \in LN(CF)} \left( \bigwedge_{OS_{pre} \in pre(CF_1)} OS_{pre} \right) \right) \bigcup \left( \bigwedge_{i \in LN(CF)} \left( \bigwedge_{OS \in TOS(CF_1)} OS \right) \right)
\]

Fig. 31. Example for Overlapped Combined Fragments

\[\mu_{CF} = \bigwedge_{m \in TOP(CF)} \left( \bigwedge_{OS_p \in \text{Init}(m)} \neg OS_p \bigcup \bigwedge_{OS_q \in \text{TOS}(m)} OS_q \right)\]

Fig. 32. Example for Combined Fragments with Interaction Constraints

\section*{F. Discussion}

This paper does not address timed events, \emph{i.e.}, the events can not represent the occurrence of an absolute time. The Messages are disallowed to cross the boundaries of CFs and their Operands [32]. Thereby, gates are not discussed in this paper. We only handle complete Messages, each of which has both sending and receiving OSs. The lost and found Messages are out of the scope of this paper.

For nested CFs, our syntactical constraints restrict that the borders of any two CFs can not overlap each other, \emph{i.e.}, the inner CF can not cover more Lifelines than the outer CF. The example in figure 31 is ill-formed. In this way, Coregion can only contain OSs and Coregions, no other CFs can be enclosed within a Coregion.

An Interaction Constraint of an Operand is located on a Lifeline where the first OS occurring within the Operand, \emph{i.e.}, the Interaction Constraint is positioned above the first OS occurring within the Operand. For example, figure 1b contains a Parallel covering three Lifelines. In the first Operand of the Parallel, \emph{op1}, either OS \emph{s2} or OS \emph{s3} may be the first OS to execute. As the Interaction Constraint of \emph{op1}, \emph{cond1}, located on Lifeline \emph{L2}, OS \emph{s2} executes before OS \emph{s3}.

However, if an Interaction Constraint of an Operand is located above a nested CF, it may not restrict an OS to be the first one to execute. In the example of figure 32, Interaction Constraint \emph{cond1} is located above a Parallel, which expresses that the first OS occurring within the Option’s Operand is contained by the Parallel on Lifeline \emph{L2}. However, OS \emph{s1} and OS \emph{s2}, which may be the first one to execute within the Parallel, are located on \emph{L1}. To avoid the contradiction, we assume that an Interaction Constraint can restrict an OS to be the first one to execute only if it is located above an OS, not a nested CF.

For each Operand whose Constraint evaluates to \emph{True}, the order between the first OS occurring within the Operand and any other OSs which are directly enclosed in the Operand is captured by an LTL formula (see figure 33). Function \emph{Init(m)} returns the first OS occurring within Operand \emph{m}, which may return an empty set if the Interaction Constraint is located above a nested CF.

\section*{VI. Verifying Safety and Consistency Properties}

Practitioners tend to construct multiple Sequence Diagrams to capture the requirements and design of a system. A Sequence Diagram may present possible executions, describing how the environment and system interact with each other, or specify desired property of the system. For the former case, we consider (in the previous section) that all the traces derived from a Sequence Diagram are complete traces. For the latter case, we adopt partial trace semantics to define the Sequence Diagram since the OS traces derived from it can be interleaved by OSs of Messages that do not appear in the property’s Sequence Diagram but appear in an system execution, modeled by Sequence Diagram. In this section we present how to generate safety and consistency properties as LTL formulas from Sequence Diagrams with Negative and Assertion respectively.

\subsection*{A. Safety Property with Negative Combined Fragment}

A Negative CF defines a set of invalid traces. While creating a collection of Sequence Diagrams to specify a system’s behavior, we wish to ensure that the system is safe in the
sense that none of the invalid traces exist. We define two types of safety properties: the weak safety property and the strong safety property. If there is no trace of the system containing any invalid trace of a Negative as a sub-trace, we consider the system is weak safe with respect to the Negative. A system is strong safe with respect to a Negative if any trace of the system does not contain the OSs which are ordered as an invalid trace of the Negative, i.e., the invalid traces can be interleaved. The strong safety properties focus on the order of OSs of invalid traces, which can be specified as an LTL template \( \Omega_{seq}^{\text{SNCF}} \),

\[
\Omega_{seq}^{\text{SNCF}} = \neg(\Phi^{CF} \land \delta_i) \quad (\text{where formula } \Phi^{CF} \text{ asserts the order of OSs enclosed in the Negative, and } \delta_i \text{ enforces the interleaving semantics of partial traces of the Negative, i.e., at most one OS of the Negative can execute at once; other OSs which do not appear in the Negative but appear in an system execution may occur and interleave the partial traces.)}
\]

We define a temporal logic template, \( \Omega_{seq}^{\text{WNCF}} \), to characterize the weak safety property of Sequence Diagram \( seq \) with respect to a Negative. Formally,

\[
\Omega_{seq}^{\text{WNCF}} = \neg(\Phi^{CF} \land \delta_i) \quad (\text{where formula } \Phi^{CF} \text{ asserts the order of OSs enclosed in the Negative, and } \delta_i \text{ enforces the interleaving semantics of partial traces of the Negative, i.e., at most one OS of the Negative can execute at once; other OSs which do not appear in the Negative but appear in an system execution may occur and interleave the partial traces.)}
\]

### B. Consistency Property with Assertion Combined Fragment

We define that a collection of Sequence Diagrams is consistent with respect to a Sequence Diagram with an Assertion only if any trace in the Assertion on a Lifeline always follows OSs, which may happen right before the Assertion on the same Lifeline. Formula \( \Omega_{seq}^{\text{ASCFS}} \) in figure 36 represents the consistency property of \( seq \) with respect to an Assertion on Lifeline \( i \), two conditions should be satisfied if all the OSs in the set of \( \text{pre} \) happen. (1) No other OSs in Sequence Diagram \( seq \) are allowed to happen until all the OSs in the Assertion complete their execution. (2) The order among OSs within the Assertion, the Weak Sequencing between the Assertion and its preceding/succeeding Interaction Fragments, and the interleaving semantics of the Assertion’s partial traces (enforced by \( \epsilon_{\text{ASCFS}}^{\text{part}} \)) should be preserved. If an Assertion contains other CFs, the order of OSs within each nested CF \( CF_j \) on each Lifeline \( i \) is represented by \( \Phi^{CF_j} \downarrow \), which is the restriction of \( \Phi^{CF_j} \) on Lifeline \( i \). Generally, \( \Phi^{CF_j} \downarrow \) is a conjunction of sub-formulas \( \alpha, \gamma \), and additional sub-formulas (optional and various for different CFs) on Lifeline \( i \). To obtain \( \Phi^{CF_j} \downarrow \), we need to project the syntactic constructs of \( CF_j \) on Lifeline \( i \), and then keep the sub-formulas of \( \Phi^{CF_j} \) which are related to these constructs.

Based on this consistency definition, we can verify if a Sequence Diagram, \( seq \), satisfies the consistency constraints set by another Sequence Diagram with an Assertion. For example, the Sequence Diagram in figure 1a does not satisfy the consistency property specified by the Sequence Diagram in figure 35. For trace \( [s1, r1, s2, r2, s3, r4, s6] \), the mandatory trace \( [s1, r1, s2, r2] \) is not strictly included as a sub-trace.

### C. Deadlock Property with Synchronous Messages

A deadlock can occur within a Sequence Diagram with synchronous Messages, where each synchronous Message must have an explicit reply Message (see example in figure 37). We want to detect if multiple Lifelines are blocked, waiting on each other for a reply.
asynchronous Messages to express that if a Lifeline sends a synchronous Message, it can not send or receive any other synchronous Message until it receives a reply Message. We define some helper functions, where \(typeOS(\text{OS}_p)\) returns that \(\text{OS}_p\) is a sending OS or a receiving OS, and \(\text{Reply}(\text{OS}_p)\) returns the reply Message of a synchronous Message containing \(\text{OS}_p\). Constraint \(\xiaaaaaaaaaaaa\) can be used as a property to check if a Sequence Diagram is deadlock, in the sense that if all the reply Messages are received, \(\text{i.e.}\), the receiving OSs of the reply Messages have executed, no Lifeline is blocked and deadlock does not happen in the Sequence Diagram. In the example of figure 37, all of the Lifelines eventually deadlock since they all send Messages and are all awaiting replies.

\textbf{D. Ignore and Consider within Properties}

All the CFs other than Ignore and Consider CFs can be embedded within a Negative or an Assertion CF to represent properties, which can be defined as LTL formulas by replacing formula \(\xi_{\text{seq}}\) of the LTL template for Combined Fragments (see figure 5) with formula \(\xi_{\text{seq}}^\text{part}\). However, if a Negative or an Assertion CF contains an Ignore or a Consider CF to express properties, the formula of Ignore or Consider CF needs to be rewritten. The Messages that are ignored in such a CF may interleave not only the sub traces of the CF (as we define in section V), but also interleave the (partial) property traces. We need to define an LTL formula to address this issue. The LTL formula of Ignore within properties constrains the semantics of Messages appearing in the Ignore with formulas \(\theta_{\text{CF}}\) and \(\gamma_{\text{CF}}\) (see figure 39). For Consider, the OSs of considered message types which do not appear in the Consider cannot occur to interleave the partial sub traces of the CF, which is captured by formula \(\zeta_{\text{ConsiderOs}(\text{CF}) \setminus \text{AOS}(\text{CF})}\) (see figure 40). Function \(\text{ConsiderOs}(\text{CF}) \setminus \text{AOS}(\text{CF})\) returns the OSs of the considered message types which do not appear in \(\text{CF}\), where \(\text{ConsiderOs}(\text{CF})\) returns the set of OSs of the considered message types.

\(\Psi_{\text{Consider}} = \theta_{\text{CF}} \land \bigwedge_{i \in \text{LN}(\text{CF})} \gamma_{\text{CF}}^i \land \zeta_{\text{ConsiderOs}(\text{CF}) \setminus \text{AOS}(\text{CF})}\)

\(\zeta_S = \square \left( \bigwedge_{\text{OS}_n \in \Sigma} (\neg \text{OS}_n) \right)\)

\(\xi_{\text{seq}} = \bigwedge_{i \in \text{LN}(\text{CF})} (\diamond \left( \bigwedge_{\text{OS}_p \in \text{pre}(\text{CF}_i)} (\oslash \text{OS}_p) \rightarrow \left((\bigwedge_{\text{OS}_n \in (\text{AOS}(\text{seq}_1) \setminus \text{AOS}(\text{CF}_i))} (\neg \text{OS}_n) \right) \left( \bigwedge_{\text{OS}_r \in \text{AOS}(\text{CF}_i)} (\oslash \text{OS}_r) \right) \right) \right) \land \bigwedge \left( \bigwedge_{\text{CF}_j \text{needed}(\text{CF})} (\Phi_{\text{CF}_j}^{\text{part}} \text{part})) \right)\)

\(\Phi_{\text{CF}_j}^{\text{part}} = \left( \bigwedge_{g \in \text{BERU}(\text{CF}_i)} \alpha_g \right) \land \left( \bigwedge_{\text{CF}_j \text{nested}(\text{CF})} \left( \left( \bigwedge_{\text{CF}_j \text{part}(\text{CF})} \Phi_{\text{CF}_j}^{\text{part}} \right) \right) \right)\)

\(\Phi_{\text{CF}_j}^{\text{part}} = \left( \bigwedge_{g \in \text{BERU}(\text{CF}_i)} \alpha_g \right) \land \left( \bigwedge_{\text{CF}_j \text{nested}(\text{CF})} \left( \left( \bigwedge_{\text{CF}_j \text{part}(\text{CF})} \Phi_{\text{CF}_j}^{\text{part}} \right) \right) \right)\)

\(\sqrt{\text{fig. 36}}\). Consistency property for Assertion

\(\sqrt{\text{fig. 40}}\). LTL formula for Consider in Property

\(\text{VII. VERIFYING SEQUENCE DIAGRAMS VIA NU\text{SMV}}\)

In the previous sections, we present the semantics of Sequence Diagrams with CFs as LTL formulas. This formalization enables a user to express certain properties using Sequence Diagrams. We hypothesize that such a framework also forms the basis for a practitioner to use a decision procedure including model checking as a means to verify her Sequence Diagrams. In this section, we examine this hypothesis by developing techniques and tools to translate Sequence Diagrams into the input language of NuSMV (a model checking tool), allowing us to enumerate all possible traces expressed by a Sequence Diagram and verify properties specified using Negative and Assertion CFs. Additionally, we can evaluate the conformance of the two separate representations, helping us gain higher-level assurance of our transformation processes. Moreover, our approach enables the verification of Sequence Diagrams against desired temporal properties provided by software engineers.

In practice, some Sequence Diagrams for a single system are intended to express some valid executions, while others are to express that certain executions are invalid (using Negative CFs) or mandatory (using Assertion CFs). We translate Sequence Diagrams for modeling valid behavior into NuSMV modules and others into LTL formulas respectively. Then, NuSMV can be leveraged to enumerate system runs of a Sequence Diagram and determine whether the Sequence Diagram is safe \(\text{i.e.}, the set of “intended” valid traces and the set of invalid traces are disjoint and consistent \(\text{i.e.}, the set of valid traces is a subset of the mandatory traces)\).

This section first presents the evaluation of Interaction Constraints using system runs. Next, we provide an overview of NuSMV features that are sufficient for a reader to understand our approach, followed by an overall mapping strategy from Sequence Diagram to the input language of NuSMV. Then, we illustrate how to translate a basic Sequence Diagram (or a CF’s Operand that does not contain other CFs) into NuSMV modules. Finally, we show how to translate all types of CFs and nested CFs into NuSMV modules.

\(\text{A. System Run vs Trace}\)

A Sequence Diagram for a system expresses possible traces \(\text{of event occurrences). The complete behavior of the system is specified as a set of runs, each of which is a sequence of system states. A run can be finite, denoted as } \rho = \rho_0 \rho_1 \ldots \rho_n, \text{ or infinite, denoted as } \rho = \rho_0 \rho_1 \ldots \rho_n \ldots \text{ Each } \rho_i \in \mathcal{Q} \text{ is a system state, } \rho_0 \text{ is an initial state, and } \rho \text{ may end with a final state } \rho_n \text{ if the run is finite. } R(\rho_i, \sigma_{i+1}, \rho_{i+1}) \text{ is a transition from state } \rho_i \text{ to } \rho_{i+1} \text{ upon taking event occurrence } \sigma_{i+1}. \text{ Given a trace}}\)
of event occurrences $\sigma \in \Sigma^\infty$, we define that the run $\rho \in Q^\infty$ induced by $\sigma$ as a sequence of states inductively if it exists: $\rho_0$ is the initial state of $\rho$ and $R(\rho_0, \sigma_1, \rho_1)$. $(\sigma_0$ does not exist.) For each $i \in \mathbb{N}$, if $R(\rho_i, \sigma_{i+1}, \rho_{i+1})$, then $R(\rho_{i+1}, \sigma_{i+2}, \rho_{i+2})$.

To check if a system run expressed by the NuSMV model is induced by a trace of events expressed in LTL, we need to additionally consider the evaluation of the Constraints of CFs. In Sequence Diagram $\text{seq}$, an OS $\sigma_i$, may take place if the Constraints of the CFs enclosing $\sigma_i$ evaluates to True, denoted as a set of boolean expressions $\text{cond}(\sigma_i)$. Recall that CFs can be nested, $\text{cond}(\sigma_i)$ contains not only the Constraint of the immediate CF $CF_i$, but also the Constraints of all the CFs that enclose $CF_i$. Given a trace $\sigma$ and a run $\rho$, they are compatible with respect to $\text{seq}$ if and only if for each $\sigma_i \in \Sigma_{\text{seq}}$, where $\Sigma_{\text{seq}}$ is the set of all OSs of $\text{seq}$, every Constraint in $\text{cond}(\sigma_i)$, evaluates to True in $\rho_i$, i.e., $\bigwedge_{c \in \text{cond}(\sigma_i)} [c]_{\rho_i} = \text{True}$.

Definition 7 We define that $\sigma$ compatibly induces $\rho$ if and only if $\sigma$ and $\rho$ are compatible and $\rho$ is induced by $\sigma$.

To check if a Sequence Diagram satisfies the desired LTL properties, we translate the Sequence Diagram into NuSMV modules, expressing all possible system runs of the Sequence Diagram. The properties are either specified by practitioners directly using LTL or generated from Negative or Assertion CFs using our LTL framework. With the assistance of the NuSMV model checking tool, we can verify the Sequence Diagram, represented as the NuSMV modules, against those properties. Similarly, we can represent the same Sequence Diagram in terms of both LTL and NuSMV model, to determine if the two representations are compatible.

B. NuSMV Overview

NuSMV is a model checking tool, which exhaustively explores all executions of a finite model to determine if a temporal logic property holds. For a property that does not hold, a counterexample is produced showing an error run. A NuSMV model consists of one main module without formal parameters and may include other modules with formal parameters. An instance of a module can be created using the VAR declaration within another module to create a modular hierarchy. To access variables of instance modules, the instance name with . (DOT) can be used followed by the variable name. The composition of multiple modules can be parallel or interleaving. If the modules are indicated as process modules, they are interleaved in the sense that exactly one of the modules (including main) executes in each step.

NuSMV modules are finite state machines (FSMs). Variables must be of finite types or module instances, declared inside each module. The initial states are defined by using an \texttt{init} statement of the form $\text{init}(\text{x}) := \text{EXP}$, which defines the value or set of values $\text{x}$ can assume initially. Transitions are represented by using the \texttt{next} statements of the form $\text{next}(\text{x}) := \text{EXP}$, which defines the value or set of values that $\text{x}$ can assume in the following state. All the transitions in a module execute concurrently in each step. Derived variables (i.e., macros) are defined by using assignment statements of the form $\text{x} := \text{EXP}$ and they are replaced by \text{EXP} in each state. The system’s invariant is represented with the INVAR statement, which is a boolean expression satisfied by each state.

C. Mapping Overview

We base the mapping of a Sequence Diagram to the input language of NuSMV on syntactic deconstruction and the formal semantics given by our logical framework. A Sequence Diagram is represented as the main module. We map the Lifelines into respective NuSMV modules, which are instantiated and declared in the main module. Recall that a CF is projected onto each of its covered Lifelines to obtain a CEU. Accordingly, its Operand on each of the covered Lifelines forms an EU. Both CEUs and EUs are represented as NuSMV modules.

Each CEU is declared as a module instance, which we call a submodule in its Lifeline module. To enforce that multiple CEUs at the same level on each Lifeline adhere to their graphical order, we define a derived variable, \texttt{flag final}, for each CEU module, to indicate whether the CEU completes its execution. A CEU is composed of one or more EUs, each of which is instantiated as a submodule inside the CEU module. The execution order of multiple EUs (i.e., the transfer of control among them) is determined by the Interaction Operator that composes them into the CEU (the translation of each Operator is discussed later in this section). In the case that a Sequence Diagram contains nested CFs (i.e., a CEU consisting of an EU that encloses other CEUs), we map each enclosed CEU as a submodule of the containing EU’s module. This procedure is recursively applied until all CEUs and EUs are mapped accordingly.

Within Lifeline or EU modules, a directly enclosed OS is represented as a boolean variable, which initializes to \texttt{False} (note that a CEU module does not contain OS variables). Once an OS occurs, its value is set to \texttt{True} and then to \texttt{False} in the following states. This value transition expresses the fact that an OS can occur only once in the Sequence Diagram. To record the execution of OSs, we introduce an enumerated variable, \texttt{state}, in each Lifeline or EU module. \texttt{state} assumes an enumeration element to express that respective OSs have taken place and an initial value, \texttt{sinitt}, to express that no OSs have occurred yet. A CEU module contains one boolean
variable, \textit{cond}, for each of its EUs to represent the Interaction Constraint of the EU.

To express the interleaving semantics among Lifelines, we introduce an \textbf{INVAR} statement in the main module to assert that at most one OS on one of the Lifelines can take place in each step. A boolean variable \textit{chosen} is used for each Lifeline to restrict that: (1) a Lifeline is chosen only if it is enabled, \textit{i.e.}, there is an OS that is ready to take place on the Lifeline, represented by the derived variable \textit{enabled}; (2) either only one Lifeline can be chosen to execute an OS in each step if Lifelines are enabled (\textit{i.e.}, before all OSs on the Lifelines have occurred); or no Lifeline can be chosen when all Lifelines are not enabled and all \textit{chosen} variables remain \textit{False} thereafter. A sending OS is enabled to execute if and only if the OSs prior to it on the same Lifeline have already occurred. A receiving OS is enabled for execution if and only if the OSs prior to it on the same Lifeline and the sending OS of the same Message have already occurred. To execute the OSs enclosed in CFs, the variable \textit{chosen} for each Lifeline is passed to the CEU and EU modules on that Lifeline as a parameter.

\textbf{D. Basic Sequence Diagram with asynchronous Messages}

In this subsection, we illustrate our mapping strategy with an example basic Sequence Diagram as shown in figure 1a. Figure 41 shows the NuSMV description of the example, which contains a main module for the Sequence Diagram. We map the three Lifelines to three modules, which are instantiated as submodules \text{L1}, \text{L2}, and \text{L3} in the main module. We show the implementation of module \text{L2} here. Module \text{L2} takes modules \text{L1}, \text{L3} as parameters. Three OSs on Lifeline \text{L2} are defined as boolean variables \text{OS}_r1, \text{OS}_r2, and \text{OS}_r3 in the \text{VAR} section. We define each OS as \text{OS}_sx or \text{OS}_rx, where \text{s} and \text{r} denote they are sending or receiving OSs, and \text{x} is the corresponding Message name. The enumerated variable \textit{state} has four values, including a initial value \textit{sinit} and three values to record the execution of the three respective OSs. A derived variable \textit{enabled} for each OS represents the enabling condition of the OS by using the variable \textit{state} in the \textbf{DEFINE} section. For instance, \textit{r3_enabled} for OS \textit{OS}_r3 is \textit{True} if and only if the sending OS of Message \textit{m3} and the preceding OS, \textit{OS}_r2, on Lifeline \textit{L2} have taken place, \textit{i.e.}, \text{state} on Lifeline \text{L2} sets to \text{r2} and \textit{state} on Lifeline \text{L3} sets to \text{s3}. The Lifeline \text{L2} can be enabled if and only if one of \text{r1}, \text{r2}, and \text{r3} is enabled. The variable \textit{flag_final} checks whether the last OS \text{r3} on \text{L2} takes place \textit{i.e.}, \text{state} sets to \text{r3}. If so, all OSs in module \text{L2} have occurred. The \textbf{ASSIGN} section defines the transition relation of module \text{L2}. For example, \text{OS}_r3 is set to \textit{False} initially. When it is chosen and enabled, it is set to \textit{True}. It is set to \textit{False} in the subsequent states to represent that an OS can execute exactly once. Variable \textit{state} is set to \text{r1} in the same state in which \text{OS}_r1 occurs.

\textbf{E. Basic Sequence Diagram with synchronous Messages}

The translation of a Sequence Diagram with synchronous Messages is similar to the translation of a Sequence Diagram with asynchronous Messages, except that the sending Lifeline

\begin{verbatim}
MODULE main
VAR
  l_L1: L1(l_L2, l_L3);
  l_L2: L2(l_L1, l_L3);
  l_L3: L3(l_L1, l_L2);

INVAR
  {{(l_L1.chosen -> l_L1.enabled)
   & (l_L2.chosen -> l_L2.enabled)
   & (l_L3.chosen -> l_L3.enabled)}
   & ((l_L1.chosen & !l_L2.chosen & !l_L3.chosen)
     | (!l_L1.chosen & l_L2.chosen & l_L3.chosen)
     | (!l_L1.chosen & !l_L2.chosen & l_L3.chosen)
     | (!l_L1.chosen & l_L2.chosen & !l_L3.chosen)
     | (!l_L1.chosen & !l_L2.chosen & !l_L3.chosen))}

MODULE L2(L1, L3)
VAR
  OS_r1 : boolean;
  OS_r2 : boolean;
  OS_r3 : boolean;
  state : {sinit, r1, r2, r3};
  chosen : boolean;

DEFINE
  r1_enabled := state = sinit & L1.state = s1;
  r2_enabled := state = r1 & (L3.state = s2
    | L3.state = s3);
  r3_enabled := state = r2 & L3.state = s3;
  enabled := r1_enabled | r2_enabled | r3_enabled;

ASSIGN
  init(state) := sinit;
  next(state) :=
    case
      state = sinit & next(OS_r1) :r1;
      state = r1 & next(OS_r2) :r2;
      state = r2 & next(OS_r3) :r3;
      1 :state;
    esac;
  init(OS_r1) := FALSE;
  next(OS_r1) :=
    case
      chosen & r1_enabled :TRUE;
      OS_r1 :FALSE;
      1 :OS_r1;
    esac;
  init(OS_r2) := FALSE;
  next(OS_r2) :=
    case
      chosen & r2_enabled :TRUE;
      OS_r2 :FALSE;
      1 :OS_r2;
    esac;
  init(OS_r3) := FALSE;
  next(OS_r3) :=
    case
      chosen & r3_enabled :TRUE;
      OS_r3 :FALSE;
      1 :OS_r3;
    esac;

Fig. 41. Basic Sequence Diagram with asynchronous Messages to NuSMV
\end{verbatim}
blocks until a reply Message is received. We introduce a boolean variable, \textit{isBlock}, for each Lifeline to capture this semantic aspect. All OSs on a Lifeline include \textit{isBlock} as part of their enabling conditions, thus preventing the OSs from occurring while \textit{isBlock} is True.

Figure 42 represents the NuSMV description of a Sequence Diagram with synchronous Messages (see figure 37), containing a module for Lifeline \textit{L1}. Each OS name is prefixed with either \textit{sync} for a synchronous Message, or \textit{reply} for a reply Message. Each OS has an enabling condition \textit{!isblock} indicating that the OS can not be enabled when the Lifeline is blocked. \textit{isBlock} initializes to \textit{False} and is set to \textit{True} when the sending OS \textit{sync_s1} executes. It is set to \textit{False} when the OS \textit{reply_r1} of a reply Message arrives and the execution of other OSs resumes. Note that portions of the module definition have been excluded that are redundant with the module definition for a basic Sequence Diagram with asynchronous Messages.

MODULE \textit{L1} (L2, L3)
VAR
OS\_sync\_s1:boolean;
OS\_sync\_r3:boolean;
OS\_reply\_r1:boolean;
OS\_reply\_s3:boolean;
state: \{sinit, sync\_s1, sync\_r3, reply\_r1, reply\_s3\};
chosen :boolean;
isblock:boolean;
DEFINE
sync\_s1\_enabled := state = sinit & !isblock;
sync\_r3\_enabled := state = sync\_s1 & !isblock & ( L3.state = sync\_s3 | L3.state = reply\_s2 | L3.state = reply\_r3);
...
ASSIGN
...
init(OS\_sync\_s1) := FALSE;
next(OS\_sync\_s1) := case
chosen & sync\_s1\_enabled :TRUE;
OS\_sync\_s1 :FALSE;
1 :OS\_sync\_s1;
esac;
init(OS\_sync\_r3) := FALSE;
next(OS\_sync\_r3) := case
chosen & sync\_r3\_enabled :TRUE;
OS\_sync\_r3 :FALSE;
1 :OS\_sync\_r3;
esac;
...
init(isblock) :=FALSE;
next(isblock) := case
next(OS\_sync\_s1) & !next(OS\_reply\_r1):TRUE;
next(OS\_reply\_r1) :FALSE;
1 :isblock;
esac;
...

Fig. 42. Basic Sequence Diagram with synchronous Messages to NuSMV

**F. Combined Fragments**

A CF enclosing multiple Lifelines is projected onto all the Lifelines to obtain a collection of CEUs, one for each Lifeline. A CEU contains a collection of EUs, one for each Operand on the same Lifeline. To preserve the structure of the Sequence Diagram during translation, we map a CF to NuSMV submodules, one for each Lifeline module, while the EUs are mapped to NuSMV sub-submodules of their parent CEU submodule separately. We implement the Interaction Constraint for each Operand with a boolean variable \textit{cond}. We do not control the value of \textit{cond} until the Operand is ready to enter, representing the fact that a condition may change during the execution of the Sequence Diagram. If \textit{cond} evaluates to \textit{True}, the Operand is entered, otherwise, the Operand is skipped. Afterwards, the value of \textit{cond} stays the same. While there is no Constraint in an Operand, \textit{cond} is defined as constant \textit{True}. Thus, the NuSMV implementation of Interaction Constraints is consistent with the LTL semantics of the Constraints. An extra variable \textit{op\_eva} for each Operand indicates its respective execution status, including “not ready” (the OSs that may happen prior to the Operand on the Lifeline have taken place) by enumeration element - 1, “ready but not enabled” (the Constraint evaluates to \textit{False}) by enumeration element 0, and “start” (Constraint evaluates to \textit{True}) by enumeration element 1. \textit{cond} is evaluated when the Operand is ready to be entered, i.e., \textit{op\_eva} evaluates to either 0 or 1. Both \textit{cond} and \textit{op\_eva} for each Operand are instantiated and declared in the CEU module on the Lifeline where the Interaction Constraint of the Operand is located. The value of \textit{op\_eva} is passed to other CEUs of the same CF as parameters, which is further passed to all the EUs of the same operand to coordinate multiple EUs. From the deconstruction of Sequence diagrams and CFs (see section 3), we define the OSs as boolean variables in the respective EUs that directly enclose them, instead of the CEUs; OSs that are not enclosed in any CF are declared as boolean variables in their Lifeline module.

1) \textit{Concurrent}: In a Parallel CF, the Operands are interleaved, which is captured using a strategy similar to the implementation of interleaved Lifelines modules. We introduce a boolean variable \textit{chosen} for each EU module to indicate whether the EU is chosen to execute. We add an INVAR statement for each CEU to assert that (1) either only one EU module is chosen to execute or no EUs are enabled (i.e., all EUs have completed execution or their Constraints evaluate to \textit{False}), and (2) an EU module can be chosen only if it is enabled (i.e., an OS within the EU is enabled to execute). All INVAR statements are combined using logical conjunctions to form a global invariant in the main module. An example to illustrate the translation rules is shown and explained in section VII-F2.

2) \textit{Atomic execution}: A Critical Region has a sole Operand with each CEU module having a single EU submodule. We use a boolean variable, \textit{isCritical}, for each EU of the Critical Region’s Operand, to restrict the OSs within the EU from interleaving with other OSs on the same Lifeline. Variable \textit{isCritical} is initialized to \textit{False} in each EU module of the Critical Region’s Operand. It is set to \textit{True} if the EU starts to execute OSs and stays \textit{True} until the EU finishes execution.
Once the EU completes, \emph{isCritical} is set to \emph{False}. The negation of \emph{isCritical} of an EU is considered as an enabling condition for each variable of other OSs, which may interleave the EU, on the same Lifeline. See figure 1b for an example. On Lifeline \emph{L3}, the sending OS of Message \emph{m6} takes the negation of \emph{isCritical} for the EU on Lifeline \emph{L3} as an enabling condition.

Figure 1b shows an example Sequence Diagram with nested CFs, \emph{i.e.}, a Parallel containing a Critical Region. The implementation of its main module and the modules of Lifeline \emph{L2} and its CEUs and EUs are shown in figure 43. In the module of Lifeline \emph{L2}, the Parallel’s CEU module is initialized as a module instance. Two EUs of the Parallel’s Operands are initialized as two module instances within its CEU module. The CEU module of the Critical Region is initialized in the Parallel’s EU module as a module instance and it is declared separately, which contains a module instance for the EU of the Critical Region’s Operand.

In the Parallel, the Interaction Constraint of its Operand, \emph{op1}, is located on \emph{L2}. Thus, \emph{cond1} for \emph{op1} is initialized and declared in the Parallel’s CEU module on \emph{L2}. It is set to the value of the evaluation step and remains that value in the following steps. Variable \emph{op1-ev} is initialized to \emph{-1}, and then is set depending on the value of \emph{cond1} when entering the CEU, \emph{i.e.}, it is set to \emph{1} if \emph{cond1} evaluates to \emph{True} or \emph{0} otherwise. In each EU module of the Parallel, a variable \emph{chosen} is used to denote whether the EU is chosen to execute OSs.

In the EU module of the Critical Region’s Operand, \emph{isCritical} is initialized to \emph{False} and is set to \emph{True} if OS \emph{r3} has executed, \emph{i.e.}, the EU of the Critical Region’s Operand on Lifeline \emph{L2} starts to execute. It remains \emph{True} until the EU finishes execution, and then is set to \emph{False} to allow other OSs on the same Lifeline to execute. On Lifeline \emph{L2}, each of the OSs which may interleave the execution of Critical Region’s EU, \emph{i.e.}, OS \emph{r5} and OS \emph{r6}, takes \emph{isCritical} as an enabling condition, denoting that these OSs may execute only if the control is not in the EU of the Critical Region.

3) \textbf{Branching}:

a) \textbf{Representing Alternatives}: The Alternatives maps to CEU modules, one for each Lifeline, containing EU submodules, one for each Operand. For each Operand, a boolean variable \emph{exe} indicates the execution status of the applicable Operand, \emph{i.e.}, \emph{exe} is set to \emph{True} if the Operand is chosen to execute. The variable \emph{exe} for each Operand is initialized and declared in the CEU module on the Lifeline where the Operand’s Constraint is located. The Constraint under INVAR restricts that an Operand’s \emph{exe} can be set to \emph{True} only if the Operand’s \emph{cond} evaluates to \emph{True}. It also restricts that at most one Operand can be chosen to execute, \emph{i.e.}, at most one \emph{exe} can be set to \emph{True} at a time, or all Operand Constraints evaluate to \emph{False}. The use of \emph{exe} guarantees that all the enclosed Lifelines choose the same Operand’s EU module to execute to avoid inconsistent choices (\emph{e.g.}, Lifeline \emph{L1} chooses Operand 1’s EU whereas Lifeline \emph{L2} chooses Operand 2’s EU). The \emph{cond} of the chosen Operand stays \emph{True} and those of the unchosen Operands are set to \emph{False} and stay \emph{False}.

Figure 9 is an example of an Alternatives with three
MODULE par_L2(state, L2_chosen, par_L1, par_L3)
VAR
  op1 : par_op1_L2(L2_chosen, par_L1.op1, par_L3.op1, op1_eva);
  op2 : par_op2_L2(L2_chosen, par_L1.op2, par_L3.op2, par_L1.op2_eva, state, op1.CF2.op1.isCritical);
  cond1 : boolean;
  op1_eva : -1..1;
DEFINE
  enabled := op1.enabled | op2.enabled;
  flag_final := op1.flag_final & op2.flag_final;
ASSIGN
  init(op1_eva) := -1;
  next(op1_eva) := case
    op1_eva=-1&next(state)=r1&!next(cond1):0;
    op1_eva=-1&next(state)=r1&next(cond1):1;
    1 :op1_eva;
  esac;
  init(cond1) := {TRUE, FALSE};
  next(cond1) := case
    op1_eva = -1 :{TRUE, FALSE};
    op1_eva != -1 :cond1;
    1 :cond1;
  esac;

Fig. 43. NuSMV module for Parallel Operands enclosing three Lifelines. Figure 45 shows the Alternatives’s CEU module on Lifeline L2. Three modules are instantiated to represent three EUs respectively. All the Interaction Constraints for the three Operands are located on Lifeline L2. Thus, the variables op_eva, cond, and exe for the three Operands are instantiated and declared in the CEU module on Lifeline L2. For example, variable op1_eva for Operand op1 initially is set to -1, and then is set depending on the values of exe of op1, i.e., it is set to 1 if exe evaluates to True, denoting op1 is chosen. Otherwise, op1_eva is set to 0 to denote op1 is unchosen. cond1 stays True if op1 is chosen, or it is set to False and stays False if op1 is unchosen. exe1 stays to the value of evaluation in the following steps. The variables of the other two Operands are defined in the same way as the ones of op1. The INVAR statement in the main module expresses the strategy of choosing at most one Operand to execute as we described.

b) Representing Option: For each Lifeline, The Option CF is mapped to a CEU module and its sole Operand is mapped to an EU module, using a similar but simpler strategy than the Alternatives. An enumerated variable, op1_eva, is used to describe the execution status of a single EU module. The variable is initialized to -1 in the CEU module as is explained in section VII-F. We demonstrate an example of an Option in figure 8. Figure 46 represents the implementation of the Option’s CEU module and its Operand’s EU module on Lifeline L2. If cond1 evaluates to True, op1_eva is set to 1 to allow the EU module to execute OSs. Otherwise, op1_eva is set to 0 to skip the EU module. Variable op1_eva is passed to the EU module as an enabling condition of the first OS, s2, in the EU. A derived variable flag_final of an EU module that evaluates to True represents that the OSs within the EU will not execute in the following steps, i.e., the OSs have executed
Fig. 44. NuSMV module for Critical Region or the EU is skipped. The rest of the EU module is the same as the Lifeline module for a basic Sequence Diagram with asynchronous Messages.

c) Representing Break: The Break has been rewritten to an Alternatives with two Operands as we describe in section V-B2. Therefore, the Break can be mapped to NuSMV modules as an Alternatives.

4) Iteration: We represent a fixed, bounded Loop with NuSMV modules, where the Loop body iterations are composed using Weak Sequencing. To unfold the Loop, each OS is mapped to an array of boolean variables, whose length is the number of iterations. The graphical order of the OSs within the same iteration is maintained, and the OSs among iterations execute sequentially along a Lifeline, \textit{i.e.}, OSs in iteration \textit{n} take place before OSs in iteration \textit{n+1}.

For example, the NuSMV module in figure 47 implements the EU of the Loop’s Operand on Lifeline \textit{L1} in figure 14, with three iterations. OSs \textit{s1} and \textit{r3} are mapped to two arrays of three boolean variables, \textit{i.e.}, the unfolded EU contains six OSs. The variable \textit{state} has a value for each OS to record its execution, \textit{i.e.}, the value of \textit{state} is set to \textit{s1}_{2}, representing OS \textit{s1} in the second iteration has taken place. Between two iterations, the first OS of the succeeding interaction takes the last OS of the preceding iteration as an enabling condition, \textit{i.e.}, OS \textit{s1}\textit{[3]} representing \textit{s1} in the third iteration, which is enabled only if \textit{r3} in the second iteration (\textit{OS}_{r3}[2]) has executed.

5) Weak Sequencing and Strict Sequencing: Mapping a Weak Sequencing or a Strict Sequencing to the input language of NuSMV obtains a CEU module for each Lifeline, which contains an EU module for each Operand. The semantics of the Weak Sequencing enforces the total order among EUs of Operands on the same Lifeline. To describe the semantics, any EU module (except the first one) takes variable \textit{flag\_final} of the preceding EU on the same Lifeline as an enabling condition, \textit{i.e.}, the EU can not execute before the preceding one completes.

Figure 20 is an example of a Weak Sequencing, whose EU of the second Operand on Lifeline \textit{L2} is mapped into a NuSMV module (see figure 48). In the EU module, the first OS, \textit{r4}, has an enabling condition, which is the variable \textit{flag\_final} of the EU occurring immediately before this EU (the EU of the first Operand). In this way, the order between these two modules on Lifeline \textit{L2} can be enforced.

The semantics of the Strict Sequencing enforces the total
Fig. 46. NuSMV module for Option

order between adjacent Operands. An EU module of an Operand (other than the first one) within a Strict Sequencing takes the variables flag_final of every EU module within the previous Operand as enabling conditions of respective OSs. It asserts that all EUs can not execute until its previous Operand completes execution.

Figure 21 is an example of a Strict Sequencing and figure 49 shows the EU module of the secondOperand on Lifeline L2. The OS r4 takes the variables flag_final, one for each EU of the firstOperand as enabling conditions to enforce the total order among Operands.

6) Ignore and Consider: Ignore and Consider make it possible to execute the Messages not explicitly appear in the CF. An Ignore specifies a list of message types which do not appear in the Ignore. The Messages of ignored types can occur and interleave the traces of the Ignore. A Consider specifies a
MODULE weak_op2_L2(chosen, weak_L1_op2, weak_L3_op2, weak_L2_op1, op2_eva)
VAR
  OS_r4 : boolean;
  state : {sinit, r4};
DEFINE
  r4_enabled := state = sinit & op2_eva = 1
  & weak_L1_op2.state = s4
  & weak_L2_op1.flag_final;
  enabled := r4_enabled;
  flag_final := (state=r4 & op2_eva=1)
  | op2_eva=0;
ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_r4) :r4;
    1 :state;
  esac;
  init(OS_r4) := FALSE;
  next(OS_r4) := case
    chosen & r4_enabled :TRUE;
    OS_r4 :FALSE;
    1 :OS_r4;
  esac;
... 
Fig. 48. NuSMV module for Weak Sequencing

MODULE strict_op2_L2(chosen, strict_L1_op2, strict_L3_op2, strict_L1_op1, strict_L2_op1, strict_L3_op1, op2_eva)
VAR
  OS_r4 : boolean;
  state : {sinit, r4};
DEFINE
  r4_enabled := state = sinit & op2_eva = 1
  & strict_L1_op2.state = s4
  & strict_L2_op1.flag_final
  & strict_L3_op1.flag_final;
  enabled := r4_enabled;
  flag_final := (state=r4 & op2_eva=1)
  | op2_eva=0;
ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_r4) :r4;
    1 :state;
  esac;
  init(OS_r4) := FALSE;
  next(OS_r4) := case
    chosen & r4_enabled :TRUE;
    OS_r4 :FALSE;
    1 :OS_r4;
  esac;
... 
Fig. 49. NuSMV module for Strict Sequencing

list of message types which should be considered within the Consider CF. It is equivalent to ignore other message types, i.e., the message types not in the list do not appear in the Consider, but they may occur. If a message type is considered but does not appear in the Consider CF, the Messages of the type can not occur within the Consider CF. For example, the Consider in figure 25 considers Messages m2, m3, and m5, but only m2 and m3 appear in the Consider. Thus, Message m5 can not occur within the Consider. To map an Ignore (Consider) into NuSMV modules, we assume the signature of any Message of ignored (considered) types is given, i.e., the Lifelines where the sending OS and receiving OS of a Message occur are known.

In a Sequence Diagram with an Ignore, each OS of any ignored Message is mapped to a boolean variable in the EU module of the Ignore on the Lifeline where it is located. An OS of any ignored Message can be enabled if it has not executed and the control is in the EU module. To record the status of each ignored Message’s OS, an enumeration type variable os_chosen is introduced, which is initially -1. It is set to 0 if the OS is chosen to execute and is set to 1 and stays 1 in the following steps. In each EU module of the Ignore, the OSs of ignored Messages and other OSs are interleaved, which is captured by INVAR statements using the same strategy as the implementation of Parallel.

Figure 24 illustrates an example with an Ignore. In the example, the EU of the Ignore on Lifeline L3 is mapped to an EU module (see figure 50). The Message m3 is ignored, whose receiving OS r3 is mapped to a boolean variable. A boolean variable r3_chosen is used to record the status of OS r3. OS r3 can be enabled if and only if it has not executed before and the sending OS of m3 has taken place.

In a Sequence Diagram with a Consider, each OS of the considered message types can be defined as a boolean variable in the EU module of the Consider on the Lifeline where it is located. If the OS does not appear in the Consider, it is defined as a derived variable, whose value is False to indicate the OS will never occur. For other known but not considered Messages, their OSs are defined in the same way as the OSs of ignored Messages. For example, figure 51 shows an EU module on Lifeline L2 for the Consider in figure 25. Message m5 is considered but does not appear in the Consider CF, so its sending OS s5 is mapped to a derived boolean variable OS_s5 whose value set to False. Message m6 is not considered in the Consider, its sending OS s6 is mapped to a boolean variable OS_s6, whose status is recorded by boolean variable s6_chosen. In each EU module of the Consider, each OS of the Messages not considered can interleave other OSs, which is represented by INVAR statements.

7) Coregion: We represent a Coregion in a similar way as the translation of Parallel. Each OS in a Coregion is considered as a Parallel Operand on a single Lifeline, and is mapped to an EU module with an OS variable, a state variable, and variable chosen.

8) General Ordering: A General Ordering enforces the order between two unordered OSs, which describes that one
MODULE ignore_op1_L3(L3_chosen, op1_L2, op1_eva)
VAR
OS_r2 : boolean;
state : {sinit, r2};
chosen : boolean;
OS_r3 : boolean;
r3_chosen : {-1, 0, 1};
DEFINE
r2_enabled := state=sinit & op1_L2.state=s2 & op1_eva=1;
enabled := r2_enabled;
flag_final := (state=r2 & op1_eva=1) | op1_eva=0;
r3_enabled := r3_chosen!=1 & op1_eva=1 & op1_L2.s3_chosen=1;
ASSIGN
init(state) := sinit;
next(state) := case
state = sinit & next(OS_r2) :r2;
1 :state;
esac;
init(OS_r2) := FALSE;
next(OS_r2) := case
chosen & L3_chosen & r2_enabled :TRUE;
OS_r2 :FALSE;
1 :OS_r2;
esac;
init(OS_r3) := FALSE;
next(OS_r3) := case
chosen & L3_chosen & s2_enabled :TRUE;
OS_r3 :FALSE;
1 :OS_r3;
esac;
init(r3_chosen) := (-1, 0);
next(r3_chosen) := case
r3_chosen = -1 :(-1, 0);
next(OS_r3) :1;
1 :r3_chosen;
esac;
...

Fig. 50. NuSMV module for Ignore

OS must occur before the other OS. General Ordering adds the preceding OS as part of the enabling condition of the succeeding OS, i.e., the succeeding OS can execute only if the preceding OS has executed.

G. Interaction Use

The specified Sequence Diagram of an Interaction Use can be considered as a CF, whose Interaction Operator is ref. The CF and the interaction fragments, which are directly enclosed by the referring Sequence Diagram, are combined using Weak Sequencing. On each Lifeline, the Interaction Use CF is mapped to a NuSMV module, which is initialized in the module of the specified Interaction. If the Interaction contains CFs, each of its CEUs is mapped to a CEU module and the EUIs within each CEU are mapped as per the strategy of that particular CF. In this way, the Interaction Use CF can be mapped to NuSMV modules recursively.

VIII. TOOL SUITE IMPLEMENTATION AND EVALUATION

As a proof-of-concept, we have developed a tool suite, implementing the techniques described in this paper. Figure 52 illustrates the architecture of our tool suite.

The software engineer creates his/her Sequence Diagrams in MagicDraw and selects a set of them as input to NuSMV via our MagicDraw plugin. Our tool accepts an LTL formula that can be specified manually or translated automatically from a Sequence Diagram. The Sequence Diagrams are transformed into a format appropriate for NuSMV and then verified automatically by the NuSMV model checker. If there are no property violations, the software engineer receives a positive response. If property violations exist, NuSMV generates a counterexample which is then passed to our Occurrence Specification Trace Diagram Generator (OSTDG) tool. The output
from the OSTDG is an easy-to-read Sequence Diagram visualization of the counterexample to help the software engineer locate the property violation faster. Thus, the software engineer may transparently verify his/her Sequence Diagrams using NuSMV, staying solely within the notation realm of Sequence Diagrams.

We evaluate our technique with a case study of ISIS (Insurance Services Information System), a web application currently used by the specialty insurance industry. Our evaluation uses two Sequence Diagram examples from the design documentation of ISIS.

A. Case Study Example 1: Adding Location Coverage

The first example addresses adding insurance coverage to a new location. Location type and tier (a hurricane exposure rating factor) asynchronously determine the coverage premium rate. The location’s tier is asynchronously determined by zip code. In order to charge the correct premium for a location’s windstorm coverage, the correct tier value must be determined before the rate is fetched. The Sequence Diagram of this example is shown in figure 53.

B. Case Study Example 2: End-of-month

The second example concerns an administrative procedure known as “end-of-month” which seals that month’s billing data and generates end-of-month reports for the insurance carrier. End-of-month can take several days and involve multiple users. During this time the client must remain free to continue to use ISIS. However, if end-of-month reporting occurs before the billing data is sealed, the reports may contain inaccurate data and create inconsistencies in future reports. The Sequence Diagram of this example is shown in figure 55.

C. Empirical Result

In our first case study example, we ascertain the possibility of obtaining an incorrect rate from the server (the safety property, which is translated from a Negative shown in figure 54). An invalid trace was discovered in the model by NuSMV, indicating that there is a possibility of incorrect rate determination. Using a counterexample visualization from the OSTDG (see figure 57), we easily located the Messages involved in the property violation. In reality, locating this bug manually
without our automatic technique involved a great deal more time and effort. Model checking the consistency property of a Sequence Diagram with Assertion (see figure 56) against example 2’s model returned true, indicating that end-of-month processing is always followed by end-of-month reporting.

We used NuSMV to check the two examples on a Linux machine with a 3.00GHz CPU and 32GB of RAM. Case Study example 1 executed in 19 minutes 49 seconds with 3,825 reachable states out of total 3.71e+012 states. Case Study example 2 executed in 18 minutes 14 seconds with 192 reachable states out of total 4.95e+012 states.

IX. RELATED WORK

To the best of our knowledge, our technique is the first to support all CFs and the nested CFs. Working towards similar goals, Kugler et al. [22] and Kumar et al. [24] have presented translations from a Live Sequence Chart (LSC) to LTL formulas, which does not support all CFs. Their approach formalizes global behaviors using synchronous Messages, while our work focuses on the execution of events on each Lifeline and supports both asynchronous and synchronous Messages, which is more specific for concurrent systems. Harel and Maoz [19] propose a Modal Sequence Diagram (MSD) to specify semantics of Negation and Assertion Operators, providing an avenue for us to define liveness and safety properties. To specify and formalize temporal logic properties, Autili et al. [4], [5] propose the Property Sequence Chart (PSC), which is an extension of UML 2 Sequence Diagrams. Their approach eases software engineers’ efforts for defining properties. Our method can be adapted for PSC to support a larger set of properties. Lamsweerde et al. [39] develop an approach for inferring goal specifications, in terms of temporal logic, which covers positive scenarios and excludes negative ones. But, they only consider simple scenarios without control constructs, such as CFs. More recently, Letier and Lamsweerde [25] provide a pattern to infer compositional pieces of incremental operational specification from declarative temporal property specifications. Whittle presents a three-level notation with formal syntax and semantics for specifying use cases in [41]. Each use case is defined by a set of UML Interactions in level-2 and the details of each Interaction are defined in level-3. With this three-level notation, Whittle and Jayaraman present an algorithm for synthesizing well-structured hierarchical state machines from scenarios [42]. The generated hierarchical state machines are used to simulate scenarios and improve readability. Our work focuses on Sequence Diagrams in level-3. Micskei and Waevelynck survey comprehensively formal semantics proposed for Sequence Diagrams by 13 groups and present the different options taken in [28]. In these groups, [21] presents an operational semantics for a translation of an Interaction into automata, which is used to model check the communication produced by UML state machines with SPIN or UPPAAL. Grosu and Smolka [17] propose a formal semantics using Büchi automata and represent positive and negative automata as liveness and safety properties respectively. The approaches of both groups do not support all CFs and the interpretation of automata restricts the specification of CF constraints. Eichner et al. introduce a compositional formal semantics of UML 2 Sequence Diagrams using colored high-level Petri Nets [14]. The semantics represents a subset of the CFs of Sequence Diagrams. Haugen et al. present the formal semantics of the UML 2 Sequence Diagram through an approach named STAIRS [20]. STAIRS provides a trace-based representation for a subset of Sequence Diagram CFs, which focuses on the specific definition of refinement for Interactions. Balancing flexibility and simplicity in expressing temporal properties, Mitchell [29] demonstrates that there is a unique minimal generalization of a race-free partial-order scenario even if it is iterative. He [30] also extends the Mauw and Reniers’ algebraic semantics for the Message Sequence Chart (MSC) to describe the UML 2 Sequence Diagram, whose deadlock property is defined differently from ours. To relate state-based behaviors with scenario-based descriptions, Bontemps et al. formally study the problem of scenario checking, synthesis, and verification of the LSC in [7]. Their work focuses on providing an algorithm and proving the complexity for each problem. Uchitel et al. [38] synthesize a behavioral specification in the form of a Finite Sequential Process, which can be checked using their labeled transition system analyzer.
Working towards similar goals, Damas et al.
synthesize a labeled transition system model from both positive and negative
scenarios, expressed in MSC [9]. They adopt the semantics
definitions from [38]. In addition, they generate temporal
properties from scenarios. Whittle and Schumann [43] develop
an approach to compose UML 1 Sequence Diagrams into
UML statecharts. Messages are annotated with pre-conditions
and post-conditions in terms of the UML Object Constraint
Language (OCL) to refine their meanings. A comprehensive
survey of these synthesis approaches and others’ work can be
found in [26]. Our work formalizes all the CFs in LTL, which
helps us to synthesize a collection of Sequence Diagrams.

Verification of scenario-based notation is well-accepted as
an important and challenging problem. Lima et al. provide a
tool to translate UML 2 Sequence Diagrams into PROMELA-
based models and verify using SPIN, with counterexample
visualizations [27]. Their translation does not include all of
the UML 2 CFs. Alur et al. [3], [1], [2] formalize MSCs
using automata. They synthesize state machines from MSCs
and detect safe realizability of missing scenarios [2]. They
also examine different cases of MSC verification of temporal
properties and present techniques for iteratively specifying
requirements [1]. They focus on MSC Graphs (an aggregation of
MSCs) and techniques for determining if a particular MSC is
realized in an MSC Graph. To model check MSCs, Alur et al.
present a formal semantics of MSCs based on automata theory
[3]. We extend their work to encompass more complicated
aggregations using CFs. Peled et al. have performed intensive
research on the verification of MSCs [31], [18], in particular,[33] present an extension of the High-Level MSC. They
specify MSC properties in temporal logic and check for safety
and liveness properties. As UML 2 Sequence Diagrams have
similar expressive features, our technique can be extended to
work with their approach. Kugler et al. improve the technique
of smart play-out, which is used to model check LSCs to avoid
violations over computations [23]. They can detect deadlock
of dependent moves while our technique can check for desired
properties. Walkinshaw and Bogdanov [40] detail an inference
technique to constrain a finite-state model with LTL. These
constraints reduce the number of traces required as input to a
model checker for discovery of safety counter examples. Our
work can automatically model check each Sequence Diagram
of a system against LTL properties separately, which helps to
alleviate the state explosion problem.

Inconsistency among design models in UML notations can be
quite problematic on large software development projects
where many developers design the same software together.
Finkelstein et al. [16] define the Viewpoints Framework:
an approach where each developer has her own viewpoint
composed only of models relevant to her. Blanc et al. [6]
address the problem of inconsistency between multiple Use
Case and requirements models by checking model construction
operations against logical inconsistency rules. They check
only for consistency problems, excluding safety and deadlock
dependencies through trace analysis among distinct model el-
ements that represent similar concepts. His method relaxes the
tedious requirement of rigorous trace management by accom-
modating uncertainty in input and can detect inconsistencies
between the models. Egyed et al. also develop approaches
Sequence, State, and Class Diagrams using a set of consistency
rules to check for well-formed syntax and coherence among
the models. Their approach is based on UML 1.3 modeling
notation and does not include more complicated features like
CFs.

X. Conclusion

In this paper, we present a novel logical framework to
formalize the semantics of Sequence Diagrams and all CFs
with LTL formulas, representing each semantic aspect as a
separate concern. This enables software practitioners to
verify if a Sequence Diagram satisfies specified properties
and if a set of Sequence Diagrams are safe and consistent.
To evaluate the framework, we represent the model using
NuSMV modules. We check the model against the consistency
and safety properties expressed by LTL formulas, which are
translated from Assertion and Negative CFs. Counterexamples
are visualized as Sequence Diagrams to help practitioners
locate violations. We supplement our technique with a proof-
of-concept tool suite and perform a case study of an industry
web application. We believe our approach can be extended
to define the semantics of variants of Sequence Diagram and
even other scenario-based languages.

References

[1] Rajeev Alur, Kousha Etesami, and Mihalis Yannakakis. Inference of
specifying temporal properties: an automated approach. Autom Softw
model inconsistency through operation-based model construction. In
Live Sequence Charts to state machines and back: A guided tour. TSE,
symbolic model checker. Int. Journal on Software Tools for Technology
[9] Christophe Dumas, Bernard Lambeau, Pierre Dupont, and Axel van
Lamsweerde. Generating annotated behavior models from end-user
[10] Alexander Egyed. Resolving uncertainties during trace analysis. In FSE,
[11] Alexander Egyed. Instant consistency checking for the UML. In ICSE,
and evaluating choices for fixing inconsistencies in UML design


APPENDIX A AUXILIARY FUNCTIONS

Our logical framework formalizes the Sequence Diagrams with CFs as LTL formulas, which evaluates the Interaction Constraints of Operands using auxiliary functions, e.g., function $AOS(CF)$ is defined to represent a set of OSs which are enabled and chosen to execute in $CF$, which can be represented as:

$$AOS(CF) = \begin{cases} 
\bigcup_{CF \in \text{nested}(CF)} \bigcup_{i \in \text{OSs}} TOS(CF) \\
\bigcup_{CF \in \text{nested}(CF)} \bigcup_{i \in \text{OSs}} TOS(CF) \\
\bigcup_{CF \in \text{nested}(CF)} \bigcup_{i \in \text{OSs}} TOS(CF) \\
\bigcup_{CF \in \text{nested}(CF)} \bigcup_{i \in \text{OSs}} TOS(CF) 
\end{cases}$$

where function $TOS(u)$ is overload to Combined Fragment or Interaction Operand $u$, $m$ is the chosen Operand if $CF$ is an Alternatives.

Functions $TOP(u), TBEU(u), TOS(u)$ and $\text{nested}(u)$ are introduced to make the templates succinct. For instance, $TBEU(u)$ can be represented as

$$\bigwedge_{g \in TBEU(\text{BEU}_1)} \bigvee_{h \in \text{AEBE}(\text{BEU}_1)} \left(\left(\text{COND}(h) \land \alpha_h\right) \lor \neg\text{COND}(h)\right).$$

We introduce functions $\text{pre}(u)$ and $\text{post}(u)$ to return the set OSs which happen right before or right after CEU $u$ in section V. The functions $\text{pre}(u)$ and $\text{post}(u)$ take the CEU $u$ and (by default) the Sequence Diagram as arguments. To calculate the $\text{pre}(u)$ of CEU $u$, we focus on the CEU or EU $v$ prior to $u$ on the same Lifeline:

- Case1: If $v$ is a BEU whose condition evaluates to $\text{True}$, $\text{pre}(u)$ returns a singleton set containing the last OS within $v$.
- Case2: If $v$ is a CEU with a single BEU whose condition evaluates to $\text{True}$ and contains no nested CEUs, $\text{pre}(u)$ returns a singleton set containing the last OS of the BEU.
- Case3: If $v$ is a CEU with multiple BEUs whose conditions evaluate to $\text{True}$ and contains no nested CEU,
  - Case3.1: $v$ with Operator “par” obliges $\text{pre}(u)$ to return a set containing the last OS of each BEU;
  - Case3.2: $v$ with Operator”alt” forces $\text{pre}(u)$ to return a singleton set containing the last OS of the chosen BEU; (We introduce a variable “exe” for $\text{BEU}$ of each Operand to indicate the chosen BEU $\bigvee_{i \in [1..m]} \text{exe}_i \land \bigwedge_{i \in [1..m]} (\text{exe}_i \rightarrow \text{cond}_i)$, where $m$ is the number of BEUs).
- Case3.3: \( v \) with Operator “weak” or “strict” makes \( \text{pre}(u) \) return a singleton set containing the last OS of the last BEU.
- Case4: If \( v \) is a CEU with EUs whose conditions evaluate to \( \text{False} \) or a BEU whose condition evaluates to \( \text{False} \), we check the BEU or CEU prior to \( v \) until a BEU or a CEU with the last one EU whose condition evaluates to \( \text{True} \) is found. \( \text{pre}(u) \) returns an empty set while there is no such BEU or CEU.
- Case5: If \( v \) is a CEU containing nested CEUs,
  - Case5.1: If \( v \) directly contains EU \( q \), which is the only EU whose condition evaluates to \( \text{True} \), we focus on \( E_u \) and the last CEU \( w \) which is directly enclosed in \( q \).
  - Case5.1.1: If there is a BEU after \( w \), which is directly enclosed in \( q \), \( \text{pre}(u) \) returns the last OS of the BEU.
  - Case5.1.2: If there is no BEU after \( w \) within \( q \), we recursively apply cases 2 to 5 by replacing \( v \) with \( w \).
  - Case5.2: If \( v \) directly contains multiple EUs whose conditions evaluate to \( \text{True} \).
    - Case5.2.1: \( v \) with Operator “par” makes us recursively apply case 1 or case 5.1 to each EU whose conditions evaluate to \( \text{True} \).
    - Case5.2.2: \( v \) with Operator “alt” makes us recursively apply case 5.1 to the chosen EU. \( \bigvee_{i \in [1..m]} \text{exe}_i \land \bigwedge_{i \in [1..m]} (\text{exe}_i \rightarrow \text{cond}_i), \) where \( m \) is the number of EUs.
    - Case5.2.3: \( v \) with Operator “weak” or “strict” makes us recursively apply case 5.1 to the last EU.

\( \text{post}(u) \) can be calculated in a similar way.

**APPENDIX B IMPLEMENTATION OF LTL TEMPLATES**

To express these auxiliary functions using LTL formulas, we need to discuss that who evaluate the Constraints, and when the Constraints are evaluated. For each Operand, its Constraint is located on the Lifeline where the first OS of the Operand will occur [32]. The Lifeline evaluates the Constraint and share its value with other Lifelines, which guarantees the consistency among multiple Lifelines. The time point for evaluating Constraints may be various based on different semantics. In this section, we provide our approach for handling Constraints with two semantics: the semantics of an individual Sequence Diagram or the semantics of one of multiple Sequence Diagrams in a system.

**An Individual Sequence Diagram**

In a Sequence Diagram with Messages not carrying parameters, the OSs do not change the values of variables. Thus, we consider the Interaction Constraints of Operands as rigid variables, which keep the same value in all states of a trace. In this way, evaluating the Interaction Constraints at the beginning of the execution of the Sequence Diagram is equivalent to evaluating them at the beginning of each CF. With this assumption, the Operands whose Constraints evaluate to \( \text{True} \) can be selected before the mapping from the Sequence Diagram to LTL formulas, i.e., only the Operands whose Constraints evaluate to \( \text{True} \) are mapped to LTL formulas. The auxiliary functions can avoid evaluating Constraints and be implemented directly, e.g., function \( \text{TOP}(u) \) returns the set of all Operands within \( u \) which are mapped to LTL formulas. Without loss of generality, we represent the Interaction Constraints as propositions. Our LTL template can also be adapted to handle Interaction Constraints as boolean expressions.

In the same way, the non-deterministic choice between multiple Operands of an Alternatives can also be made at the beginning of the execution of the Sequence Diagram. Only one Operand is chosen non-deterministically from the Operands whose Constraints evaluate to \( \text{True} \) and mapped to LTL formulas.

**Multiple Sequence Diagrams in a system**

The requirement or design of a system can be captured by multiple Sequence Diagrams which may share variables. In a Sequence Diagram, the values of Interaction Constraints may be modified by other Sequence Diagrams of the system during execution. Each Interaction Constraint of the CF’s Operands is evaluated when the CEU of the Lifeline where the Constraint is located is ready to execute. After evaluation, the value of each Constraint is preserved and applied to the execution of the OSs of the CF. In this way, the values of Constraints can be considered as fixed after entering the Combined Fragment.

We append the Interaction Constraints to each OS, which restricts that if an OS can occur, the Interaction Constraints associated with the OS must evaluate to \( \text{True} \) (see formula \( \bar{\xi}_{\text{seq}} \) in figure 58). An OS can be enclosed into multiple nested CFs, whose Interaction Constraints are associated with the OS, e.g., \( \text{cond}_m \) is the conjunction of the Interaction Constraints associated with \( \text{OS}_m \). Function \( \text{AllOS(seq)} \) replaces function \( \text{AOS(seq)} \) in all formulas, which returns all OSs within Sequence Diagram \( \text{seq} \). The formula \( \Phi^{CF} \) is modified as \( \Phi^{CF_s} \), which describes the execution of all \( \text{CF}_s \)‘s Operands. For Operand \( m \), if the Lifeline where \( m \)’s Constraint is located is ready to execute the CEU of \( \text{CF} \), i.e., the OSs, which happen right before the CEU, have finished execution, the Constraint is evaluated and stays to the value in the following states. If the Constraint evaluates to \( \text{True} \), function \( \Phi^m \) is satisfied by the Operand and function \( \Phi^{CF_k} \) is satisfied by each \( \text{CF}_k \) nested within \( m \). Otherwise, the Constraints of Operands of nested CF \( \text{CF}_k \) set to \( \text{False} \), denoting no OS within \( \text{CF}_k \) can occur.

Recall formula \( \alpha \) specifies that OSs execute in their graphical order on each Lifeline, and formula \( \beta \) specifies that sending OS must take place before receiving OS of the same Message. Both formulas apply the macro \( \neg \text{OS}_q \cup \neg \text{OS}_p \equiv \neg \text{OS}_q \cup (\text{OS}_p \land \neg \text{OS}_q) \) to establish the order between \( \text{OS}_p \) and \( \text{OS}_q \), i.e., \( \text{OS}_q \) can not execute before \( \text{OS}_p \). The macro indicates that \( \text{OS}_p \) must happen in some future state from current state, which can not be guaranteed for all states of a trace (see
To implement the macro with temporal operator $\square$, the macro is modified as $\square((\neg OS_q \cup OS_p) \lor (\diamond OS_p))$, which describes two cases: 1. $OS_q$ can not happen if $OS_p$ has not occurred; 2. $OS_p$ has happened before.

Formula $\gamma^{CF}_{i}$ establishes the order between the OSs within the CEU of $CF$ on Lifeline $i$ and their preceding/succeeding OSs if the Constraint of any $CF$’s Operand evaluates to True. Otherwise, the CEU’s preceding/succeeding OSs are connected using formula $\eta^{CF}$. Both formulas use the macro $\bigwedge_{OS_i \in s} (\neg OS_i \cup OS_p)$ to enforce the OSs of set $s$ can not happen before $OS_p$. However, the Constraints associated with $OS_p$ may be evaluated to False, i.e., $OS_p$ may not happen. Thus, the macro is modified as $\diamond OS_p \rightarrow (\bigwedge_{OS_i \in s} (\neg OS_i) \cup OS_p)$, which represents that if $OS_p$ can happen, the order is established. Function $TAllOS(u)$ returns the set of OSs of the BEUs directly enclosed in CEU $u$. Formula $\bar{\theta}^{CF}$ establishes the order between the first occurring OS and other OSs within the same Operand as we described in section V-D3.

For Alternatives, we assume all Operands evaluate their Constraints if any Lifeline where a Constraint is located is ready to execute the CEU of Alternatives, even if Constraints of Operands are located on different Lifelines. It guarantees that all Operands whose Constraints evaluate to True are ready to be chosen at the same time. To choose an Operand non-deterministically, we have introduced a boolean variable $exe$ for each Operand whose Constraint evaluates to True. The variable $exe$ states that: 1. Only the $exe$ of the chosen Operand evaluates to True. 2. The Constraints of unchosen Operands set to False. 3. If any OS within an Operand can occur, the $exe$ for the Operand evaluate to True.

Both LTL formulas of Critical Region and Assertion use sub-formula $\bigwedge_{OS_i \in M} \diamond OS_k$ to denote that all OSs within $M$ have occurred. Since some OSs may not happen, the sub-formula is modified as $\bigwedge_{OS_i \in M} (\square \neg OS_k)$, which denotes each OS within $M$ have occurred or can not occur any more.

Figure 58 shows the modified LTL formulas we have described for LTL implementation. The LTL formulas of other CFs can be modified in a similar way. We have implemented all CFs using LTL formulas in our tool.
$$\Pi_{seq} = \bigwedge_{i \in LN(seq)} \bigg( \bigwedge_{g \in ABEU(seq_{1,i})} \bigg( \bigwedge_{j \in MSG(seq)} \beta_j \bigwedge_{\Phi_{CF} \in \text{Anested}(seq)} \Phi_{CF} \bigwedge \bar{\epsilon}_{seq} \bigg) \bigg)$$
$$\bar{\epsilon}_{seq} = \bigvee_{\theta_{seq} \in AllOS(seq)} \bigg( \bigwedge_{OS_{m} \in AllOS(seq)} \bigg( \bigvee_{\theta_{seq} \in AllOS(seq)} \bigg( \bigwedge_{OS_{m} \in AllOS(seq)} \bigwedge_{\theta_{seq} \in AllOS(seq)} \bigwedge_{\Phi_{CF} \in \text{Anested}(m)} \Phi_{CF} \bigg) \bigg) \bigg)$$
$$\Phi_{CF} = \bigwedge_{m \in OPND(CF)} \bigg( \bigwedge_{\Phi_{pre} \in pre(CF_{1}, L_m)} \bigg( \bigwedge_{\Phi_{post} \in post(CF_{1}, l_m)} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg)$$
$$\tilde{\beta}_{i} = \bigg( \neg RCV(j) \bigwedge \neg SN(d) \bigg) \vee \bigg( \neg SN(d) \bigg)$$

\[ \tilde{\gamma}_{CF} = \bigwedge_{i \in LN(CF)} \bigg( \bigwedge_{\Phi_{pre} \in pre(CF_{1}, l_m)} \bigg( \bigwedge_{\Phi_{post} \in post(CF_{1}, l_m)} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg) \]

\[ \tilde{\eta}_{CF} = \bigwedge_{i \in LN(CF)} \bigg( \bigwedge_{\Phi_{pre} \in pre(CF_{1}, l_m)} \bigg( \bigwedge_{\Phi_{post} \in post(CF_{1}, l_m)} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg) \]

\[ \tilde{\mu}_{CF} = \bigwedge_{m \in OPND(CF)} \bigg( \bigwedge_{\Phi_{pre} \in pre(CF_{1}, l_m)} \bigg( \bigwedge_{\Phi_{post} \in post(CF_{1}, l_m)} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg) \]

\[ \tilde{\phi}_{assert} = \bigwedge_{i \in LN(m)} \bigg( \bigwedge_{\lambda_{i, seq} \in seq_{1,i}} \bigg( \bigwedge_{\Phi_{pre} \in pre(seq_{1,i}, m_{1})} \bigg( \bigwedge_{\Phi_{post} \in post(seq_{1,i}, m_{1})} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg) \bigg) \]

\[ \tilde{\delta}_{critical} = \bigwedge_{i \in LN(m)} \bigg( \bigwedge_{\lambda_{i, seq} \in seq_{1,i}} \bigg( \bigwedge_{\Phi_{pre} \in pre(seq_{1,i}, m_{1})} \bigg( \bigwedge_{\Phi_{post} \in post(seq_{1,i}, m_{1})} \Phi_{pre} \bigwedge \Phi_{post} \bigg) \bigg) \bigg) \]

Fig. 58. LTL Formulas for Implementation of Templates