A FORMAL FRAMEWORK FOR ANALYZING SEQUENCE DIAGRAM

APPROVED BY SUPERVISING COMMITTEE:

Jianwei Niu, Ph.D., Chair

Jeffery von Ronne, Ph.D.

Weining Zhang, Ph.D.

Dakai Zhu, Ph.D.

Ram Krishnan, Ph.D.

Accepted:

Dean, Graduate School

DEDICATION

This dissertation is dedicated to my mother and father, who always support and inspire me with their unconditional love.

A FORMAL FRAMEWORK FOR ANALYZING SEQUENCE DIAGRAM

by

HUI SHEN, M. SC.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2013

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Jianwei Niu for her patience and inspiration. This dissertation could not be accomplished without her expertise and guidance. I would also like to thank all the members of my committee, Dr. Jeffery von Ronne, Dr. Weining Zhang, Dr. Dakai Zhu, and Dr. Ram Krishnan for the time they have devoted to reading my work, and their helpful comments and suggestions on this dissertation. Special thanks goes to my colleagues, Mr. Mark Robinson, Mr. Omar Haider Chowdhury, and Mr. Rocky Slavin. These talented fellows helped me work with the dissertation and related projects. Finally, I would like to thank the Department of Computer Science at the University of Texas at San Antonio and all the people who helped and supported me during my research and life.

This Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Doctoral Dissertation attest to the accuracy of this statement.

May 2013

A FORMAL FRAMEWORK FOR ANALYZING SEQUENCE DIAGRAM

Hui Shen, Ph.D.
The University of Texas at San Antonio, 2013

Supervising Professor: Jianwei Niu, Ph.D., Chair

Graphical representations of scenarios, such as UML Sequence Diagrams, 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 Sequence Diagram permit different types of control flows, including interleaving, alternative, and loop, for representing complex and concurrent behaviors. These fragments increase a Sequence Diagram's expressiveness, yet introduce a challenge to comprehend what behavior is possible in the traces that express system executions. Furthermore, software practitioners tend to use a collection of Sequence Diagrams to express multiple usages of a software system. It can be extremely difficult to determine manually that multiple Sequence Diagrams constitute a consistent, correct specification.

This dissertation introduces an approach to codify the semantics of Sequence Diagrams with Combined Fragments in terms of Linear Temporal Logic (LTL) templates. In each template, different semantic aspects are expressed as separate, yet simple LTL formulas that can be composed to define the semantics of all the Combined Fragments. In addition, we develop an approach to transform Sequence Diagrams with Combined Fragments into the input language of model checker NuSMV. The analytical powers of model checking can be leveraged to automatically determine if a collection of Sequence Diagrams is consistent. Another benefit of this approach is the ability to specify certain safety properties of a system as intuitive Sequence Diagrams.

We have developed tools to translate Sequence Diagrams to both LTL and NuSMV's input language to demonstrate that they can be automatically verified. We validate our techniques by analyzing two design examples taken from an insurance industry software application. We also model Health Insurance Portability and Accountability Act of 1996 (HIPAA) Privacy Rule using Sequence Diagrams to show that high-level policies can be described using Sequence Diagrams.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

The software development community is adopting models as a viable practice to improve the productivity and quality of software systems. Models focus on the important features of a system by abstracting away nonessential details, providing a foundation of detecting errors in the early stages of software development. In particular, we are interested in scenario-based notations that are well-accepted by software practitioners for graphically depicting interactions among software systems and their environment.

The general acceptance of a scenario-based notation, in particular, Sequence Diagram of Unified Model Language (UML), can be attributed to its relatively intuitive nature and the ability to describe partial behaviors (as opposed to model-based notations, such as statecharts and process algebras, that often represent the complete behaviors of a system or its individual component). However, the semantics of the Sequence Diagram and its control constructs, Combined Fragments, is not formally defined compared to their precise syntax description, making it extremely difficult to understand what execution traces can be derived from the Sequence Diagram. Even with a formal semantics, subtle synchronization and communication errors introduced in the Sequence Diagrams would be difficult and time-consuming to detect manually.

The objective of this dissertation is to gain theoretical understanding of the Sequence Diagram, so as to integrate formal analysis techniques with Sequence Diagrams, combining their strengths and avoiding their weaknesses to increase the accessibility of formal methods to software practitioners. We develop a formal framework to capture the semantic concerns of Sequence Diagrams, including interaction among system components and environmental actors via messages, interleaving, Combined Fragments, and nesting Combined Fragments, as separate linear temporal logic (LTL) definitions, respectively. The specifics of each Combined Fragments and variants can be expressed as additional constraints. These smaller definitions can be composed using logical conjunction to codify the complete semantics of a Sequence Diagram variant. One of the key benefits of representing Sequence Diagrams in LTL is the ability to specify certain system properties or policies as intuitive Sequence Diagrams.

1.1 Sequence Diagram with Combined Fragments

Sequence Diagrams focus on the message interchange among multiple entities. In UML 1, a Sequence Diagram is typically used to express a single scenario, which represents an usage using a sequence of message exchange. UML 2 adds several major features to the Sequence Diagram, such as Combined Fragments and Interaction Use, in order to allow multiple scenarios to aggregate in a single Sequence Diagram. Combined Fragments permit different types of control flow for presenting concurrent behaviors. For instance, a Combined Fragment can represent a choice of multiple behaviors (Alternatives Combined Fragment), an interleaving composition among multiple behaviors (Parallel Combined Fragment), an atomic behavior (Critical Region Combined Fragment), or iterations of a behavior (Loop Combined Fragment). One Sequence Diagram can refer to another Sequence Diagram (copying the contents of the referred Sequence Diagram) via Interaction Use.

Combined Fragments largely increase the expressiveness of Sequence Diagram. However, precisely interpreting and analyzing Sequence Diagrams with Combined Fragments, is challenging. The semantics of Combined Fragments is described in terms of sets of valid and invalid traces by OMG, but it is not formally defined how to derive the traces compared to their precise syntax descriptions [55].

Further, Combined Fragments can be nested, providing more combinations of control flows. For instance, 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 instance, a Combined Fragment representing a critical region on each enclosing Lifeline may be nested within a Parallel Combined Fragment representing interleaving control flows. The lack of formal semantics of Sequence Diagrams makes it difficult to comprehend what behavior is possible in the traces that express system executions.

1.2 Linear Temporal Logic and Model Checking

Temporal logics express dynamic behaviors which are changing in time [58]. LTL [37] is a temporal logic, specifying the orders of events and states using temporal operators and logical connectives. It models an execution path as an infinite sequence of states or events. As a decision procedure for LTL, model checking [16] is an automatic technique for verifying reactive system, which is represented as a finite model. It exhaustively explores all possible executions of the model to determine if the model satisfies a desired property, which can be expressed using an LTL formula. If the model satisfies the property, an answer $true$ is shown. Otherwise, a counterexample is given to demonstrate an error execution.

1.3 Problem Statement

Specifying and analyzing the behaviors of a single system using multiple Sequence Diagrams with Combined Fragments is a challenging task for several reasons. First, the semi-formal semantics of Sequence Diagrams, especially of (nested) Combined Fragments, makes it difficult for practitioners to understand and use them to precisely model software systems. Next, software practitioners can construct multiple Sequence Diagrams that represent complementary perspectives of a single system. Determining that these Sequence Diagrams provide a consistent specification manually can therefore be extremely difficult. Finally, although there exist automated verification tools, which can verify whether a behavior model satisfies desired properties, there is a mismatch between Sequence Diagrams and input language of these tools.

1.4 Related Research Efforts

Many researches have proposed their approaches using different languages, including temporal logic [40, 42], automata [31, 34, 39], Petri nets (colored Petri nets) [27, 29], PROMELA [47], and template semantics [64], to provide a formal semantics for scenario-based notations. Micskei and Waeselynck [51] survey and categorize 13 approaches of UML Sequence Diagram semantics. As one of the earliest approach, Storrle [66] proposed a trace-based semantics of the UML 2 Sequence Diagram, introducing the semantics of all 12 Combined Fragments. Motivated by analyzing scenarios based requirements, Kugler et al. [40] and Kumar et al. [42] have described the semantics of LSC using temporal logic. 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 [34] propose a Modal Sequence Diagram (MSD), which is an extension of the UML 2 Sequence Diagram based on the universal/existential concepts of LSC. Their approaches increase the expressive power of the Sequence Diagram to specifying liveness and safety properties. They mainly consider synchronous Messages and Interaction Fragments are combined using Strict Sequencing. Grosu and Smolka [31] propose a formal semantics of the 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. Haugen et al. present the formal semantics of the UML 2 Sequence Diagram through an approach named STAIRS [35]. STAIRS provides a trace-based representation for a subset of Combined Fragments, focusing on the specific definition of refinement for Interactions. To specify and formalize temporal logic properties, Autili et al. [8, 9] propose the Property Sequence Chart (PSC), which is an extension of UML 2 Sequence Diagrams. Their approach eases software engineers' efforts for defining properties. Most of the work does not cover the semantics of all the Combined Fragments, in particular, nested Combined Fragments, Interaction Constraints, and both synchronous and asynchronous messages.

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. [30] define the Viewpoints Framework: an approach where each developer has her own view-point composed only of models relevant to her. Blanc et al. [11] address the problem of safety and consistency between multiple use case and requirements models by checking model construction operations against logical inconsistency rules. Egyed [23] proposes a method for identifying model dependencies through trace analysis among distinct model elements that represent similar concepts. Egyed et al. also develop approaches [24] [25] [26] to detect and repair inconsistencies between 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 Combined Fragments.

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 [47]. Their translation does not support Critical Region, Strict Sequencing, Negative, Assertion, Consider, Ignore Combined Fragments, synchronous Messages and Interaction Constraint. Van Amstel et al. present four complementary approaches for analyzing UML 1.5 Sequence Diagrams, which do not support Combined Fragments [69]. They model check Sequence Diagrams using SPIN. Alawneh et al. introduce a unified paradigm to verify and validate prominent UML 2 diagrams, including Sequence Diagrams, using NuSMV [2]. Their approach supports Alternatives and Parallel Combined Fragments. To model check MSCs, Alur et al. [6, 7] formalize MSC using automata. They examine different cases of MSC verification of temporal properties and present techniques for iteratively specifying requirements [5]. They focus on MSC Graph, which is an aggregation of MSCs. We extend their work to encompass more complicated aggregations using Combined Fragments. Peled et al. perform intensive research on the verification of MSCs [32, 54], in particular, they present an extension of the High-Level MSC [57]. They specify MSC properties in temporal logic and check for safety and liveness properties. Kugler et al. improve the technique of smart play-out, which is used to model check LSCs to avoid violations over computations [41]. They can detect deadlock of dependent moves while our technique can check for desired properties. Most of the previous work does not cover the semantics of all the Combined Fragments.

1.5 Approach

Thesis Statement: The main goal of this work is to provide a formal framework which formalizes Sequence Diagrams with Combined Fragments using LTL formulas and NuSMV model. It enables users to automatically verify multiple Sequence Diagrams are consistent. It can also express high-level properties and policies using Sequence Diagrams.

To help us use and analyze Sequence Diagram with Combined Fragments, we have developed a formal framework to represent its semantics in LTL, as LTL is a natural choice for specifying traces. We use LTL formulas to express the semantic aspects prescribed by Sequence Diagram constructs, each of which defines the execution orders among events. We deconstruct Sequence Diagrams and Combined Fragments to obtain fine-grained syntactic constructs, and provide a collection of simple LTL definitions to represent the separate aspects of the semantics to conquer the complexity of Combined Fragments. The semantics that is common to Sequence Diagrams and the 12 Combined Fragments is captured as a template, which is a conjunction of those simpler definitions. The specifics of Combined Fragments 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. Our technique supports all Combined Fragments, the nested Combined Fragments, both asynchronous and synchronous Messages, and Interaction Constraints. 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. We provide the proofs of correctness for the LTL templates, *i.e.*, the LTL templates capture the semantic aspects of Sequence Diagram with Combined Fragments.

Our approach bridges the gap between intuitive Sequence Diagrams and formal methods, increasing the accessibility of formal verification techniques to practitioners. We choose a verification tool, NuSMV [15], which is a symbolic model checker and close to industrial systems standards. We devise an approach to codify the semantics of Sequence Diagrams and Combined

Fragments in the input language of NuSMV with the help of deconstruction. We formally describe each Combined Fragment in terms of NuSMV modules. The generated NuSMV model preserves the structure of the Sequence Diagram. We provide the proofs of correctness for the NuSMV model, *i.e.*, the NuSMV model captures the semantic aspects of Sequence Diagram with Combined Fragments. To the best of our knowledge, our technique is the first to support all Combined Fragments and the nested Combined Fragments.

The Assertion and Negative Combined Fragments of Sequence Diagram describe the mandatory and forbidden behaviors respectively. Using the LTL templates, we translate the Assertion and Negative Combined Fragments into LTL specifications to express 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. We wish to ensure the system is safe in the sense that (1) all the valid traces of the system satisfy the mandatory properties represented using Assertion Combined Fragments, and (2) none of the system traces satisfy the forbidden properties represented using Negative Combined Fragments. Thus, we can verify that a set of Sequence Diagrams is safe against particular properties without requiring users to specify the LTL properties directly.

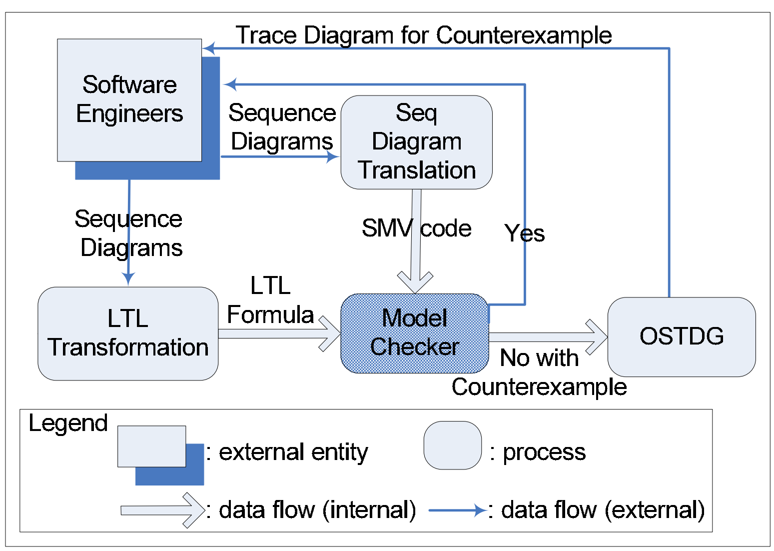


Figure 1.1: Architecture of tool suite

We have developed a proof-of-concept tool suite to implement all of the techniques. Figure 1.1 illustrates the architecture of the tool suite. We have validated our technique by analyzing and discovering violations in two design examples taken from an insurance industry software application. We have also created an Occurrence Specification Trace Diagram generator that automatically produces Sequence Diagram visualizations from NuSMV-produced counterexamples. This automation will increase the accessibility of our approach by allowing software engineers to remain focused in the realm of Sequence Diagrams.

1.6 Utility of Sequence Diagrams

As a graphical notation, Sequence Diagram is more intuitive, and easier to understand than logical expressions or textual representations for users without expertise. In the previous section, we have discussed that Sequence Diagrams can express safety properties to ease user's effort for verifying a software system. In this section, we demonstrate that Sequence Diagrams can be used to express the security requirements, especially privacy policies.

Nowadays, the storage and transmission of personal information via large-scale networks such as the Internet, may cause serious risks. Such as personal information can be used for identity theft, stalking and luring vulnerable individuals, stealing financial assets, achieving political advantage, intimidation and blackmail. Privacy policy, which is a statement or a legal document, regulates the use and disclosure of personal information. Understanding and specifying privacy policies is difficult for users and organizational policy writers without enough experiences. Sequence Diagram, which models dynamic behaviors among system actors and their environment through message passing, is an appropriate candidate for modeling privacy policies.

HIPAA (Health Insurance Portability and Accountability Act of 1996) [1] is the national standard for electronic health care transactions. It consists of general administrative requirements, administrative requirements, security rule, and privacy rule. We are interested in HIPAA privacy rule, which focuses on regulating the transmission and use of confidential health information, referred as protected health information (PHI) among covered entities. Covered entities are the organizations required to comply with HIPAA, including hospitals, insurance companies, doctors and so on. DeYoung et al. have formalized portion of HIPAA privacy rule, which sets limits and conditions on the use and disclosure of PHI using a privacy logic [21]. We model the transmission-related HIPAA privacy policies using Sequence Diagrams, which can be translated into LTL formulas via our tool suite. Our approach assists users to understand the HIPAA privacy policies. We believe that it also helps the organizational policy writers and users to verify whether their policies or the transmissions of electronic health information comply with HIPAA privacy policies.

1.7 Contributions

The main contribution of our research is six-fold:

- This dissertation proposes a technique to represent the semantics of Sequence Diagrams with Combined Fragments using LTL, including nested Combined Fragments, Interaction Constraints, and both synchronous and asynchronous messages. It also can be used to formalize the semantic variations of Sequence Diagrams [61, 64].

- The formal framework enables users to specify high-level objectives, including safety properties, and policies [59, 61].

- We also propose a technique for translating a Sequence Diagram with Combined Fragments into a NuSMV model to verify whether the Sequence Diagram meets desired properties [63].

- We develop a tool suite to implement above techniques and visualize the NuSMV counterexamples with Sequence Diagrams to ease user efforts to locate the violations [63].

- We provide the proofs of the correctness that LTL representation and NuSMV model for a Sequence Diagram correspond to the Sequence Diagram's semantic rules respectively [61].

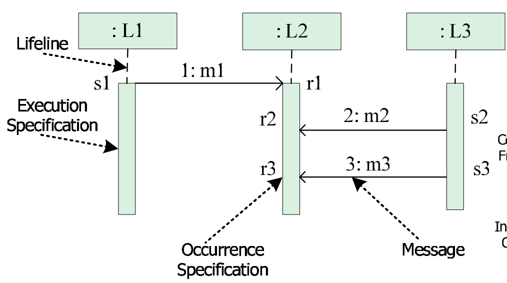
- We model HIPAA privacy rule using Sequence Diagrams to help user gain a better understanding of privacy policies, and validate our techniques and tool suite [61, 65].

1.8 Outline

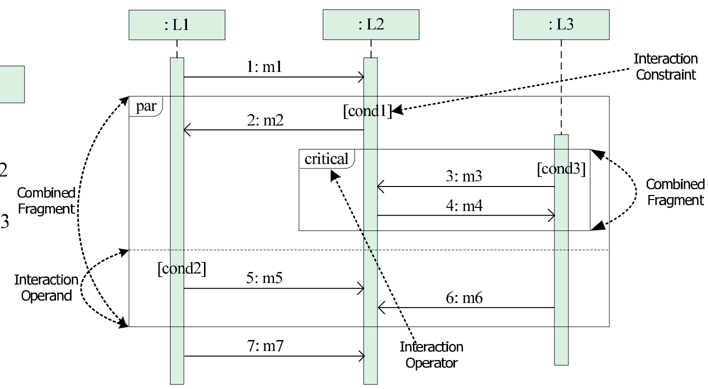
The remaining chapters of this dissertation are structured as follows. Chapter 2 summarizes the syntax and semantics of Sequence Diagrams, and presents the deconstructions of Sequence Diagram to facilitate the semantic definition. Chapters 3 discusses the trace semantics for LTL formulas and NuSMV models. Chapter 4 describes the LTL templates to represent the semantics of Sequence Diagrams with Combined Fragments. Chapter 5 discusses using Negative and Assertion Combined Fragments to express the LTL safety properties. Chapter 6 describes the formal representation of Sequence Diagrams with Combined Fragments in terms of NuSMV modules. Chapter 7 introduces our framework for automated analysis of Sequence Diagrams and describes the implementation of our tool suite. Chapter 8 validates our approach via a case study of an insurance industry software application and modeling HIPAA privacy policies using Sequence Diagrams. Chapter 9 presents related work. and we conclude with chapter 10.

Chapter 2: UML 2 SEQUENCE DIAGRAM DECONSTRUCTION

In this section, we outline the syntax and semantics of a Sequence Diagram with Combined Fragments provided by OMG [55], and present the formal definitions of a Sequence Diagram. First, we introduce the basic Sequence Diagram. Then, we discuss the structured control constructs, including Combined Fragments and Interaction Use. Next, we give a textual representation of a Sequence Diagram. Last, we deconstruct a Sequence Diagram and Combined Fragments into fine-grained syntactic constructs to facilitate the semantic description of Sequence Diagram, in particular, Weak Sequencing among Occurrence Specifications and Combined Fragments.



a. Basic Sequence Diagram



b. Sequence Diagram with Combined Fragment

Figure 2.1: Sequence Diagram syntax

2.1 Basic Sequence Diagram

We refer to a Sequence Diagram without Combined Fragments as a basic Sequence Diagram (see figure 2.1a 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 2.1a. 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 [51, 55]

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 2.1a. 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 [55].

2.2 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 2.1b 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 2.1b 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.*, $Val(seq) \cap Inval(seq) =$. 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 semantic rules general to all CFs, in the sense that, these rules do not constrain each other.

1. OSs and CFs, are combined using Weak Sequencing (defined below). On a single Life-

line, a CF's preceding Interaction Fragment must complete the execution prior to the CF's execution, and the CF's succeeding Interaction Fragment must execute subsequently.

2. Within a CF, the order of the OSs and CFs within each Operand is maintained if the Constraint of the Operand evaluates to $True$; otherwise, (*i.e.*, the Constraint evaluates to $False$) the Operand is excluded.

3. The CF does not execute when the Constraints of all the Operands evaluate to $False$. Thus, the CF's preceding Interaction Fragment and succeeding Interaction Fragment are ordered by Weak Sequencing.

The semantics of each CF Operator determines the execution order of all the Operands. Each Operator has its specific semantic implications regarding the execution of the OSs enclosed by the CF on the covered Lifelines as described in the next section.

2.3 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 non-deterministically 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 not be interleaved with any other OSs on the same Lifeline.

- **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.

2.4 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 2.1. *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 2.2. *A Message has the form* $\langle$ name, mform, OS_s, OS_r $\rangle$*, 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_r *denotes its receiving OS. Each OS has the form* $\langle l_i, loc_k, type\rangle$*, where* l_i *denotes its associated Lifeline,* loc_k *is the location where the OS takes places on Lifeline* l_i*, and* type *denotes it is either a sending or a receiving OS.*

Each Lifeline $l_i \in L$ has a set of finite locations $LOC(l_i) \subseteq \mathbb{N}$ on it. The locations form a finite sequence *1, 2, 3, ..., k,* $k \in \mathbb{N}$. 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 2.1b, the set *LOC(*l_2*)* contains seven locations, each of which is associated with an OS, *i.e.*, OSs $r1, s2, r3, s4, r5, r6, r7$. Message *msg1* is expressed by $\langle m_1, asynch, s_1, r_1\rangle$, 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 2.3. *A CF* CF_m *has the form* $\langle$ L, 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*.* oper *denotes the Interaction Operator of* CF_m*.* OP *denotes the sequence of Interaction Operands within* CF_m*,* i.e.*,* op_{m_1}*,* op_{m_2}*,...,*op_{m_n}*.*

Each $op_n \in OP$ has the form $\langle$ *L, MSG, FG, cond* $\rangle$, where *L* denotes the set of Lifelines enclosed by op_n; *MSG* denotes the set of Messages directly enclosed in op_n; *FG* denotes the set of CFs directly enclosed in op_n; and *cond* denotes the Interaction Constraint of op_n, which is *True* if there is no Interaction Constraint. Without loss of generality, *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 2.1b as an example. Sequence Diagram *seq* is represented by $\langle$ *{*l_1, l_2, l_3*}*, *{*msg_1, msg_7*}*, *{*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, msg_1, msg_7, and the set of CFs directly enclose in *seq* contains one CF, CF_1. msg_1, CF_1, and msg_7 are combined using Weak Sequencing. CF_1 is represented by $\langle$ *{*l_1, l_2, l_3*}*, *par*, *{*op_1, op_2*}* $\rangle$, where l_1, l_2, l_3 are Lifelines enclosed by CF_1, *par* is the Interaction Operator of CF_1, and op_1, op_2 are the Interaction Operands of CF_1. op_1 and op_2 preserve their execution order if their Interaction Constraints evaluate to *True* respectively, and the execution order between op_1 and op_2 are decided by Interaction Operator *par*. If both Constraints of op_1 and op_2 evaluate to $False$, CF_1 is excluded and Messages msg_1 and msg_7 are ordered by Weak Sequencing. Operand op_1 expresses the Messages and CFs directly enclosed in it, represented by $\langle$ *{*l_1, l_2, l_3*}*, *{*msg_2*}*, *{*CF_2*}*, *cond1* $\rangle$, where $cond_1$ is op_1*'s* Interaction Constraint. In this way, the syntax of *seq* is described recursively.

2.5 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 [27] 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 [55]. 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 $cf_m \uparrow_{l_i}$. (The large dotted rectangle on Lifeline *L1* in figure 2.2 shows an example).

Definition 2.4. *A CEU is given by a three tuple* $\langle$ l_i, oper, setEU $\rangle$*, where* l_i *is the Lifeline, onto which we project the CF,* oper *is the Interaction Operator of the CF, and* setEU *is the set of execution units, one for each Operand* op_n *enclosed in the CF on Lifeline* l_i*.*

Every Operand 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 $op_n \uparrow_{l_i}$. 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 2.2 shows an example of a BEU and the large dotted rectangle shows an example of an HEU).

Definition 2.5. *A BEU* u *is given by a pair,* $\langle E_u$, cond $\rangle$*, in which* E_u *is a finite set of OSs on Lifeline* l_i *enclosed in Operand* op_n*, which are ordered by the locations associated with them, and* cond *is the Interaction Constraint of the Operand.* cond *is* True *when there is no Interaction Constraint.*

Definition 2.6. *An HEU is given by* $\langle$ setCEU, setBEU, cond $\rangle$*, where* setCEU *is the set of CEUs directly enclosed in the HEU,* i.e.*, the CEUs nested within any element of* setCEU *are not considered.* 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, *cond*. (The dotted rectangle on Lifeline *L1* in figure 2.1b 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.

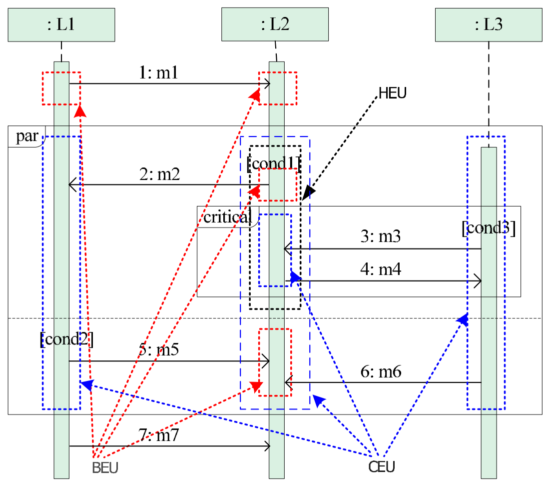


Figure 2.2: Sequence Diagram deconstruction

In the example of figure 2.1b, 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 2.3. An EU can be a BEU or an HEU, and one or more EUs compose a CEU. An HEU contains one or more CEUs.

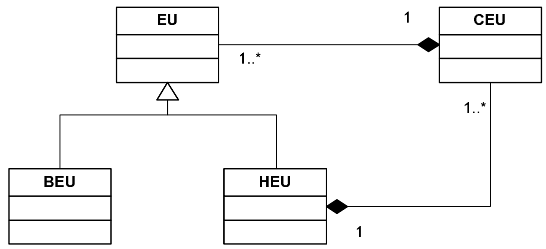


Figure 2.3: Execution Unit metamodel

2.6 Nested Combined Fragments

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 Lifelines, 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.

Chapter 3: TRACE SEMANTICS

In this chapter, we discuss the relation between the trace semantics of Sequence Diagram and the trace semantics of LTL formulas. We also build the system runs to enable the verification of property traces against the system model.

3.1 Sequence Diagram Trace vs LTL Trace

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). For Sequence Diagram seq, $(\Sigma_{sem}^{seq})^*$ represents the set of traces derived from it based on its semantic rules, where Σ_{sem}^{seq} is the set of OSs of seq. $\Sigma_{sem}^{seq} \subseteq \Sigma$, where Σ is the universe of event occurrences. The concatenation of trace υ and traces σ is represented as $\upsilon \cdot \sigma$. 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 [60]. A trace derived from a Sequence Diagram can be finite, denoted as $\upsilon[1..n] = \upsilon_1\upsilon_2...\upsilon_n$. 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 $\upsilon = \upsilon_1\upsilon_2...\upsilon_n....$

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 τ after the last OS υ_n

of each trace [31], where τ is an invisible event occurrence which does not occur in the Sequence Diagram. For instance, for a given Sequence Diagram, $seq, \forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^*, \upsilon$ is extended to $\upsilon \cdot \tau^\omega$ without changing the meaning of seq, where $\tau \in (\Sigma \setminus \Sigma_{sem}^{seq})$. Then, LTL formulas can express these traces. For instance, $(\Sigma_{LTL}^{seq})^\omega$ represents all infinite traces that satisfy the LTL representation of seq, where $\Sigma_{LTL}^{seq} = \Sigma_{sem}^{seq} \cup \{\tau\}$.

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 consider 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 safety properties 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. $\bigcirc p$ means formula p holds in the next state. $\ominus p$ means that formula p holds in the previous state. $\diamondminus p$ means that formula p holds in some past state. $\widehat{\diamondminus} p \equiv \diamondminus \ominus p$ means that formula p holds in some past state, excluding current state. $p\,\mathcal{U}\,q$ 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 $p\,\widetilde{\mathcal{U}}\,q \equiv p\,\mathcal{U}(q \wedge p)$ states that in the state when q becomes $True$, p stays $True$.

3.2 System Run vs Trace

A Sequence Diagrams expresses only example event traces of system execution. The complete behavior of a system is specified as a set of runs, each of which is a sequence of system states. A run can be finite, denoted as $\rho = \rho_0\rho_1...\rho_n$, or infinite, denoted as $\rho = \rho_0\rho_1....$ ρ_0 is an initial state, and ρ may end with a final state ρ_n if the run is finite. Each $\rho_i \in Q^\omega$ is a system state. $R(\rho_i, \sigma_{i+1}, \rho_{i+1})$ is a transition from state ρ_i to ρ_{i+1} upon taking event occurrence σ_{i+1}. Given a sequence of event occurrences $\sigma \in \Sigma^\omega$, where Σ is a set of event occurrences, we define the run ρ induced by σ as a sequence of states inductively if it exists. ρ_0 is the initial state of ρ 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 is induced by a trace of a Sequence Diagram, we need to additionally consider the evaluation of the Constraints of CFs. In Sequence Diagram seq, an OS σ_i, may take place if the Constraints of the CFs enclosing σ_i evaluates to $True$, denoted as a set of boolean expressions $cond(\sigma_i)$. Recall that CFs can be nested. $cond(\sigma_i)$ contains not only the Constraints of the immediate CF CF_i, but also all the CFs that enclose CF_i. Given a trace σ and a run ρ, they are compatible with respect to seq if and only if for each $\sigma_i \in \Sigma^{seq}$, where Σ^{seq} is the set of all OSs of seq, the Constraints of $cond(\sigma_i)$ evaluate to $True$ in ρ_i, *i.e.*, $\bigwedge\limits_{c \in cond(\sigma_i)} [\![c]\!]_{\rho_i} = True$.

Definition 3.7. *We define that σ compatibly induces ρ if and only if σ and ρ are compatible and ρ is induced by σ.*

In order to verify a Sequence Diagram expressing possible system runs, we translate it into NuSMV modules. Our LTL framework generates the possible system traces represented by the Sequence Diagram. First, using model checking technique, we can check if the traces induce the runs of the same Sequence Diagram. Second, we can verify the NuSMV modules against safety properties represented by Negative and Assertion respectively. Finally, the NuSMV modules can be checked against desired temporal properties provided by software engineers.

Chapter 4: 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 2. 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, 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 4.1) 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 $SND(j)$ 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 2.1b, function $nested(CF1)$ returns a singleton set containing Critical Region $CF2$ if the Operand of $CF2$ 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 2.2). 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.

Table 4.1: Auxiliary functions

Function	Explantation
$LN(p)$	return the set of all Lifelines in p.
$MSG(p)$	return the set of all Messages directly enclosed in p.
$SND(j)$	return the sending OS of Message j.
$RCV(j)$	return the receiving OS of Message j.
$Reply(u)$	return the reply Message of a synchronous Message containing OS u.
$typeOS(u)$	return the type of OS u, which is a sending OS or a receiving OS.
$typeCF(u)$	return the Interaction Operator of CF u.
$TOP(u)$	return the set of Interaction Operands whose Constraints evaluate to $True$ within CF u, ***i.e.***, $\{op \mid op \in OPND(u) \wedge CND(op) = True\}$, where $OPND(u)$ returns the set of all Interaction Operands in Combined Fragment u, and $CND(op)$ returns the boolean value representing the Interaction Constraint of Interaction Operand op, which is lifted to a CF containing a sole Operand.
$nested(u)$	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.
$TBEU(u)$	for CEU or EU u, return a set of directly enclosed BEUs, whose Constrains evaluate to $True$, ***i.e.***, $\{beu \mid beu \in ABEU(u) \wedge CND(beu) = True\}$, where $ABEU(u)$ returns the set of BEUs directly contained by CEU or EU u.
$AOS(q)$	return the set of OSs which are enabled (***i.e.***, the Constraints associated with it evaluate to $True$) and chosen to execute in q.
$TOS(u)$	return the set of OSs of the BEUs directly enclosed in CEU or EU u whose Constaints evaluates to $True$, ***i.e.***, $\{os \mid beu \in TBEU(u) \wedge os \in AOS(beu)\}$
$pre(u)$	return the set of OSs which may happen right before CEU u. The set contains an OS if a BEU whose Constraint 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.
$post(u)$	return the set of OSs which may happen right after CEU u, which can be calculated in a similar way as $pre(u)$.

4.1 Basic Sequence Diagram

In this section, we provide an LTL template, and prove that it represents the semantics of a basic Sequence Diagram.

4.1.1 LTL Template of Basic Sequence Diagram

We start with defining an LTL template, called Π_{seq}^{Basic} (see figure 4.1), to represent the semantics of basic Sequence Diagram. The semantic rules for basic Sequence Diagram seq defined in section 2.1 are codified separately using formulas α_g, β_j, and ε_{seq}.

α_g focuses on the intra-lifeline behavior to enforce rules 1 and 2. Recall that when projecting a Basic Sequence Diagram seq onto its covered Lifelines, $LN(seq)$, we obtain BEU g for each Lifeline i, denoted as $seq \uparrow_i$. Each BEU g contains a trace of OSs, $\sigma[r..(r + |AOS(g)| - 1)]$, where $(r \geq 0)$ and σ_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 α_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 $\neg OS_{k+1}\ \widetilde{\mathcal{U}}\ OS_k$. The second conjunct of α_g enforces that every OS in BEU g executes only once. The semantics enforced by each α_g does not constrain each other. Thus, the intra-lifeline semantics of seq is enforced by the conjunction of α_g for each Lifeline. Similarly, the semantics rule 3 is codified by a conjunction of β_j for each message j. Formula β_j enforces that, for message j, its receiving OS, $RCV(j)$, cannot happen until its sending OS, $SND(j)$ happens. Formula ε_{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 traces should execute uninterrupted until all the OSs of seq have taken place. The trace stutters at the end with τ We define the logical operator "unique or" as "$\widehat{\bigvee}$", to denote that exactly one of its OSs is chosen. A formula with logical connectives, $\bigwedge\limits_{a_i \in S} a_i$ returns the conjunction of all the elements a_i within the set S. It returns $True$ if S is an empty set.

$$
\begin{aligned}
\Pi_{seq}^{Basic} &= \Big(\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow i}} \alpha_g\Big) \wedge \Big(\bigwedge_{j \in MSG(seq)} \beta_j\Big) \wedge \varepsilon_{seq}\\
\alpha_g &= \Big(\bigwedge_{k \in [r..r+|AOS(g)|-2]} (\neg OS_{k+1}\, \widetilde{\mathcal{U}}\, OS_k)\Big)\Big) \wedge \bigwedge_{OS_e \in AOS(g)} (\neg OS_e\, \widetilde{\mathcal{U}}\, (OS_e \wedge \bigcirc\Box\neg OS_e))\\
\beta_j &= \neg RCV(j)\, \widetilde{\mathcal{U}}\, SND(j)\\
\varepsilon_{seq} &= \Box\Big(\Big(\widehat{\bigvee_{OS_m \in AOS(seq)}} OS_m\Big) \vee \Big(\bigwedge_{OS_m \in AOS(seq)} (\widehat{\Diamond} OS_m)\Big)\Big)
\end{aligned}
$$

Figure 4.1: LTL templates for basic Sequence Diagram

4.1.2 Proof for LTL Template of Basic Sequence Diagram

We wish to prove that the LTL templates for basic Sequence Diagram capture the semantics of basic Sequence Diagram. Recall that the semantic rules of basic Sequence Diagrams have been presented in section 2.1. We begin by rewriting the LTL template Π_{seq}^{Basic} into $\widetilde{\Pi}_{seq}^{Basic}$ (see figure 4.2). We show $\widetilde{\Pi}_{seq}^{Basic}$ is equivalent to Π_{seq}^{Basic} with slightly syntactical different form.

$$
\begin{aligned}
\widetilde{\Pi}_{seq}^{Basic} &= \Big(\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow_i}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j \in MSG(seq)} \rho_j\Big) \wedge \Big(\bigwedge_{j \in MSG(seq)} \beta_j\Big) \wedge \varepsilon_{seq}\\
\tilde{\alpha}_g &= \bigwedge_{k \in [r..(r+|AOS(g)|-2)]} (\neg OS_{k+1}\, \widetilde{\mathcal{U}}\, OS_k)\\
\rho_j &= (\neg SND(j)\, \widetilde{\mathcal{U}}\, (SND(j) \wedge \bigcirc\Box\neg SND(j))) \wedge (\neg RCV(j)\, \widetilde{\mathcal{U}}\, (RCV(j) \wedge \bigcirc\Box\neg RCV(j)))\\
\beta_j &= \neg RCV(j)\, \widetilde{\mathcal{U}}\, SND(j)\\
\varepsilon_{seq} &= \Box\Big(\Big(\widehat{\bigvee_{OS_m \in AOS(seq)}} OS_m\Big) \vee \Big(\bigwedge_{OS_m \in AOS(seq)} (\widehat{\Diamond} OS_m)\Big)\Big)
\end{aligned}
$$

Figure 4.2: Rewriting LTL templates for basic Sequence Diagram

In $\widetilde{\Pi}_{seq}^{Basic}$, sub-formulas β_j and ε_{seq} keep unchanged from Π_{seq}^{Basic}. We can rewrite the sub-formula $\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow_i}} \alpha_g$ into a conjunction of $\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow_i}} \tilde{\alpha}_g$ and $\bigwedge_{j \in MSG(seq)} \rho_j$ (see figure 4.3). Sub-formula $\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow_i}} \alpha_g$ is equivalent to a conjunction of two sub-formulas (see line 1), where the first sub-formula is $\bigwedge_{\substack{i \in LN(seq)\\ g = seq\uparrow_i}} \tilde{\alpha}_g$ (see line 2), enforcing the total order of OSs in BEU g along each

Lifeline of seq. Recall that an OS is an event occurrence within the certain context, i.e., seq. The second sub-formula, $\bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} (\bigwedge_{OS_e \in AOS(g)} (\neg OS_e \; \widetilde{\mathcal{U}} \; (OS_e \wedge \bigcirc \Box \neg OS_e)))$, enforcing that, for all Lifelines, every OS along each Lifeline executes once and only once. It is equivalent to enforcing that, for each Message, its sending OS and receiving OS execute once and only once respectively (see line 2), which can be captured using sub-formula $\bigwedge_{j \in MSG(seq)} \rho_j$ (see line 3).

$$
\begin{aligned}
\bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} \alpha_g = & \bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} ((\bigwedge_{k \in [r..(r+|AOS(g)|-2)]} (\neg OS_{k+1} \, \widetilde{\mathcal{U}} \, OS_k)) \wedge (\bigwedge_{OS_e \in AOS(g)} (\neg OS_e \, \widetilde{\mathcal{U}} \, (OS_e \wedge \bigcirc \Box \neg OS_e)))) \\
= & \bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} (\bigwedge_{k \in [r..(r+|AOS(g)|-2)]} (\neg OS_{k+1} \, \widetilde{\mathcal{U}} \, OS_k)) \wedge \bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} (\bigwedge_{OS_e \in AOS(g)} (\neg OS_e \, \widetilde{\mathcal{U}} \, (OS_e \wedge \bigcirc \Box \neg OS_e))) \qquad (1) \\
= & (\bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq)} ((\neg SND(j) \, \widetilde{\mathcal{U}} \, (SND(j) \wedge \bigcirc \Box \neg SND(j))) \\
& \wedge (\neg RCV(j) \, \widetilde{\mathcal{U}} \, (RCV(j) \wedge \bigcirc \Box \neg RCV(j))))) \qquad (2) \\
= & (\bigwedge_{\substack{i \in LN(seq) \\ g = seq\uparrow_i}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq)} \rho_j) \qquad (3)
\end{aligned}
$$

Figure 4.3: Rewriting Π_{seq}^{Basic} into $\tilde{\Pi}_{seq}^{Basic}$

For a given basic Sequence Diagram, seq, with j Messages and $2j$ event occurrences (each Message has a sending event occurrence and a receiving event occurrence), Σ_{sem}^{seq} is the set of event occurrences of seq. $\Sigma_{sem}^{seq} \subseteq \Sigma$, where Σ is the universe of event occurrences. The set of valid traces, $(\Sigma_{sem}^{seq})^*$, contains finite traces derived from seq based on the semantic rules of Sequence Diagrams. Σ_{LTL}^{seq} is the set of event occurrences of LTL representation of seq, Π_{seq}^{Basic}, where $\Sigma_{LTL}^{seq} = \Sigma_{sem}^{seq} \cup \{\tau\}$. τ is an invisible event occurrence which does not occur in seq, *i.e.*, $\tau \in (\Sigma \setminus \Sigma_{sem}^{seq})$. $(\Sigma_{LTL}^{seq})^\omega$ represents all infinite traces that satisfy Π_{seq}^{Basic}. For each trace $\sigma \in (\Sigma_{LTL}^{seq})^\omega$, function $pre_i(\sigma)$ returns the prefix of length i of trace σ, i.e., $\sigma_{[1..i]}$. We lift function $pre_i(\sigma)$ to $PRE_i((\Sigma_{LTL}^{seq})^\omega)$ to apply to a set of traces. Function $PRE_i((\Sigma_{LTL}^{seq})^\omega)$ returns the set of the prefixes of the traces within $(\Sigma_{LTL}^{seq})^\omega$, where the length of each prefix must be i, *i.e.*, $PRE_i((\Sigma_{LTL}^{seq})^\omega) = \{pre_i(\sigma) | \sigma \in (\Sigma_{LTL}^{seq})^\omega\}$.

Lemma 4.8. *For a given Sequence Diagram, seq, with j Messages, if $\sigma \in (\Sigma_{LTL}^{seq})^\omega$, then σ must* *have the form, $\sigma = \sigma_{[1..2j]} \cdot \tau^{\omega}$, where $\sigma_{[1..2j]}$ contains no τ.*

Proof. If $\sigma \models \Pi_{seq}^{Basic}$, then $\sigma \models \varepsilon_{seq}$. We can directly infer from sub-formula ε_{seq} that, in σ, only one OS of seq can execute at a time, and σ should execute uninterrupted until all the OSs of seq have taken place. Similarly, we can infer from the assumption that $\sigma \models \bigwedge_{j \in MSG(seq)} \rho_j$. From sub-formula $\bigwedge_{j \in MSG(seq)} \rho_j$, we can infer that each OS within seq can execute once and only once in σ. seq contains j Messages with $2j$ OSs, so σ should have the form, $\sigma = \sigma_{[1..2j]} \cdot \tau^{\omega}$. $\square$

The semantics of a basic Sequence Diagram is given by a set of valid, finite traces, while LTL formulas describe infinite traces. To represent the semantics of a basic Sequence Diagram using LTL formulas, we need to bridge the gap by adding stuttering of τ after each finite trace of the Sequence Diagram. For instance, for a given Sequence Diagram, seq, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^*, \upsilon$ is extended to $\upsilon \cdot \tau^{\omega}$ without changing the meaning of seq.

We wish to prove that for a given Sequence Diagram, seq, with j Messages, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^*, \upsilon \cdot \tau^{\omega} \models \Pi_{seq}^{Basic}$, *i.e.*, $\upsilon \cdot \tau^{\omega} \in (\Sigma_{LTL}^{seq})^{\omega}$. The semantic rule of seq defines that each OS occurs once and only once. Thus, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^*, |\upsilon| = 2j$. From lemma 4.8, we learn that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq})^{\omega}, \sigma = \sigma_{[1..2j]} \cdot \tau^{\omega}$, where $\sigma_{[1..2j]}$ contains no τ. $\sigma_{[1..2j]} \in PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega})$. If $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^*, \upsilon \cdot \tau^{\omega} \in (\Sigma_{LTL}^{seq})^{\omega}$, we can infer that, $\upsilon \in PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega})$, *i.e.*, $(\Sigma_{sem}^{seq})^* \subseteq PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega})$.

We also wish to prove that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq})^{\omega}, \sigma_{[1..2j]} \in (\Sigma_{sem}^{seq})^*$, *i.e.*, $PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega}) \subseteq (\Sigma_{sem}^{seq})^*$.

Theorem 4.9. *For a given Sequence Diagram, seq, with j Messages, $(\Sigma_{sem}^{seq})^*$ and $PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega})$ are equal.*

We provide the proof of theorem 4.9 in appendix B.1.

4.2 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 Π_{seq}^{Basic} to Π_{seq} for expressing Sequence Diagram with CFs (see figure 4.4). We introduce a new template Φ^{CF} to assert the semantics of each CF directly enclosed in seq. Template Π_{seq} is a conjunction of the formulas α_g, β_j, Φ^{CF} and ε_{seq}, which is equivalent to the LTL template of basic Sequence Diagram if seq does not contain any CF.

$$\boxed{\Pi_{seq} = \bigwedge_{i \in LN(seq)} \Big(\bigwedge_{g \in TBEU(seq\uparrow_i)} \alpha_g \Big) \wedge \bigwedge_{j \in MSG(seq)} \beta_j \wedge \bigwedge_{CF \in nested(seq)} \Phi^{CF} \wedge \varepsilon_{seq}}$$

Figure 4.4: LTL templates for Sequence Diagram with Combined Fragments

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 α_g for each BEU g. The order of OSs within Messages directly enclosed in seq are enforced by a conjunction of β_j for each Message j. ε_{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 Φ^{CF} (see figure 4.5) 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 η^{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 η^{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 Ψ^{CF} defines the semantics of OSs directly enclosed in CF. The second conjunct states the semantics of each CF_i, which are directly enclosed in the CF is enforced by each Φ^{CF_i}. In this way, Φ^{CF} can be defined recursively until it has no nested CFs.

Template Ψ^{CF} captures the semantics that is common to all CFs (except Alternatives and Loop) (see figure 4.6). Sub-formula γ_i^{CF} 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. θ^{CF} states semantic rule 2, which defines the order among OSs directly enclosed in CF. θ^{CF} is a conjunction of α_gs and β_js. The α_gs is a conjunction of all α_g of each Lifeline, where g is a BEU whose condition evaluates to $True$. The β_js is a conjunction of β_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, Ψ_{alt}^{CF} defines the semantics of OSs and CFs directly enclosed in CF. Ψ_{alt}^{CF} and Φ^{CF_i} for CF_i nested in the Alternatives form an indirect recursion (see figure 4.11). The semantics of Loop is defined in a similar way (see figure 4.16).

Semantic rule 4 varies for CFs with different Operators, which is enforced by adding different semantics constraints on Ψ^{CF} for each individual CF respectively. The semantics specifics for different types of CF Operators are defined as below.

4.2.1 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 2.1b is an example of Parallel with two Operands. The OSs within the same Operand 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

$$\Phi^{CF} = \begin{cases} \eta^{CF} & if \quad |TOP(CF)| = 0 & (1) \\ \Psi^{CF} \wedge \bigwedge_{CF_i \in nested(CF)} \Phi^{CF_i} & if \quad (|TOP(CF)| > 0) \wedge & \\ & \quad (typeCF(CF) \neq alt) \wedge (typeCF(CF) \neq loop) & (2) \\ \Psi^{CF} & if \quad (|TOP(CF)| > 0) \wedge & \\ & \quad ((typeCF(CF) = alt) \vee (typeCF(CF) = loop)) & (3) \end{cases}$$

$$\eta^{CF} = \bigwedge_{i \in LN(CF)} ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \,\widetilde{\mathcal{U}}\, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre})))$$

Figure 4.5: LTL template for nesting Combined Fragment

$$\Psi^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF}$$

$$\theta^{CF} = \bigwedge_{i \in LN(CF)} (\bigwedge_{g \in TBEU(CF\uparrow_i)} \alpha_g) \wedge \bigwedge_{j \in MSG(TOP(CF))} \beta_j$$

$$\gamma_i^{CF} = \bigwedge_{OS \in TOS(CF\uparrow_i)} ((\neg OS \,\widetilde{\mathcal{U}}\, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre}))) \wedge ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \,\widetilde{\mathcal{U}}\, (\diamondminus OS)))$$

Figure 4.6: LTL template for OSs directly enclosed in Combined Fragment

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 (see figure 4.7).

$$\boxed{\Psi_{par}^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF}}$$

Figure 4.7: LTL formula for Parallel

4.2.2 Branching

Collectively, we call Option, Alternatives and Break Branching constructs.

Representing Option

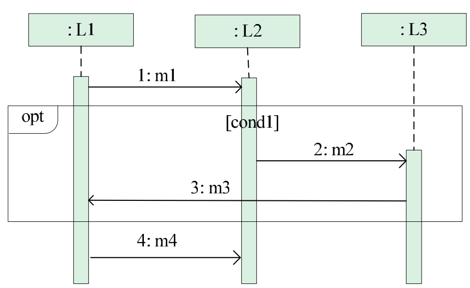


Figure 4.8: Example for OCF

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 (see figure 4.9).

$$\boxed{\Psi_{opt}^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF}}$$

Figure 4.9: LTL formula for Option

Figure 4.8 is an example of Option. The OSs within the Option execute if *cond1* evaluates to *True*. Otherwise, the Option is excluded, and its semantics is defined by formula η, *i.e.*, Messages *m1* and *m4* are combined with Weak Sequencing.

Representing Alternatives

The Alternatives chooses at most one of its Operands to execute. Each Operand must have an explicit 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 Φ_{alt}^{CF} in figure 4.11 defines the semantics of Alternatives, which is a conjunction of Φ_{alt}^{m}. Each Φ_{alt}^{m} represents the semantics of Operand m, whose Constraint evaluates to $True$, which is achieved by function $TOP(CF)$.

The semantics of the chosen Operand (if clause) is described by $\bar{\theta}_m^{CF}$, $\bar{\gamma}_{i,m}^{CF}$ and Φ^{CF_t}, where $\bar{\theta}_m^{CF}$ defines the partial order of OSs within the chosen Operand and Φ^{CF_t} defines the semantics of CFs directly enclosed in the chosen Operand. Functions Ψ_{alt}^{m} and Φ^{CF_t} 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 $\bar{\gamma}_{i,m}^{CF}$ instead of γ_i^{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 *exe* for each Operand whose Interaction Constraint evaluates to $True$. The variable *exe* should satisfy the following assertion,

$$\widehat{\bigvee_{i\in[1..m]}} exe_i \wedge \bigwedge_{i\in[1..m]} (exe_i \rightarrow cond_i)$$

The first conjunct expresses that only one *exe* sets to $True$, *i.e.*, exactly one Operand is chosen. The second conjunct enforces that the Interaction Constraint of Operand whose *exe* sets to $True$ must evaluate to $True$. Figure 4.10 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 *exe* to $True$.

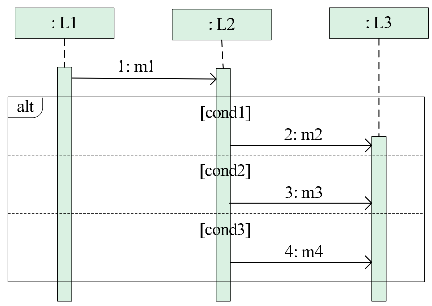


Figure 4.10: Example for Alternatives

$$\Psi_{alt}^{CF} = \bigwedge_{m\in TOP(CF)} \Psi_{alt}^{m}$$

$$\Psi_{alt}^{m} = \begin{cases} \bar{\theta}_m^{CF} \wedge \bigwedge_{i\in LN(CF)} \bar{\gamma}_{i,m}^{CF} \wedge \bigwedge_{CF_t\in nested(m)} \Phi^{CF_t} & if\ m\ is\ the\ chosen\ Operand \quad (1) \\ True & else \quad (2) \end{cases}$$

$$\bar{\theta}_m^{CF} = \bigwedge_{i\in LN(m)} \Big(\bigwedge_{g\in TBEU(m\uparrow_i)} \alpha_g\Big) \wedge \bigwedge_{j\in MSG(TOP(m))} \beta_j$$

$$\bar{\gamma}_{i,m}^{CF} = \bigwedge_{OS\in TOS(m\uparrow_i)} \Big(\big(\neg OS \big(\widetilde{\mathcal{U}} \bigwedge_{OS_{pre}\in pre(CF\uparrow_i)} (\diamondminus OS_{pre})\big)\big) \wedge \big(\big(\bigwedge_{OS_{post}\in post(CF\uparrow_i)} (\neg OS_{post})\big)\, \widetilde{\mathcal{U}}\, (\diamondminus OS)\big)\Big)$$

Figure 4.11: LTL formula for Alternatives

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 4.12). We rewrite the Sequence Diagram, using Alternatives to replace Break (see figure 4.13). $cond3$ 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.

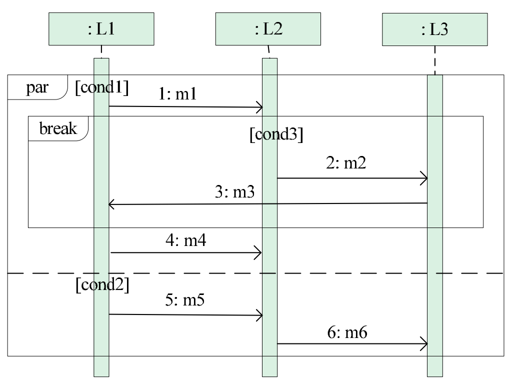


Figure 4.12: Example for Break

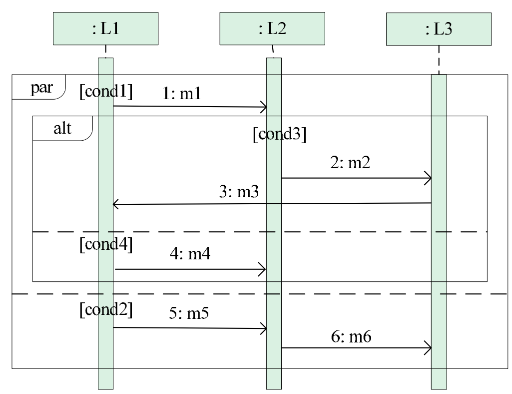


Figure 4.13: Representing Break using Alternatives

4.2.3 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 2.1b, 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 $\Psi^{CF}_{critical}$ presents the semantics for Critical Region (see figure 4.14). θ^{CF} and γ^{CF}_i have their usual meanings. δ_{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.

$$\Psi^{CF}_{critical} = \theta^{CF} \quad \wedge \bigwedge_{i \in LN(CF)} \gamma^{CF}_i \quad \wedge \bigwedge_{i \in LN(CF)} \delta_{(AOS(CF\uparrow_i),(AOS(seq\uparrow_i)\setminus AOS(CF\uparrow_i)))}$$

$$\delta_{M_1,M_2} = \Box((\bigvee_{OS_k \in M_1} OS_k) \rightarrow ((\bigwedge_{OS_j \in M_2} (\neg OS_j))\,\widetilde{\mathcal{U}}\,(\bigwedge_{OS_k \in M_1} \diamondminus OS_k)))$$

Figure 4.14: LTL formula for Critical Region

4.2.4 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. Bounded Loop, whose *maxint* is given, can be formalized using LTL formulas. First, we consider fixed Loop. Figure 4.15 is an example of fixed Loop which iterates exactly three times.

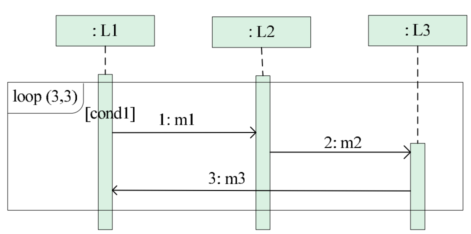


Figure 4.15: Example for Loop

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 $Loop[n]$. For example, the Loop in figure 4.15 has three iterations, *Loop [1]*, *Loop [2]* and *Loop [3]*. Figure 4.16 shows an LTL formula for a Loop. $\widehat{\theta}_R$ overloads θ^{CF}, which asserts the order of OSs during each iteration. $\widehat{\gamma}_{i,R}$ 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 nth 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.

$$\Psi^{CF}_{loop,R} = \widehat{\theta}_R \wedge \bigwedge_{i\in LN(CF)} \widehat{\gamma}_{i,R} \wedge \bigwedge_{i\in LN(CF)} \kappa_{i,R} \wedge \bigwedge_{\substack{n\in[1..R]\\ CF_t\in nested(CF)}} \Phi^{CF_t[n]}$$

$$\widehat{\theta}_R = \bigwedge_{i\in LN(CF)} \Big(\bigwedge_{g\in TBEU(CF\uparrow_i)} \widehat{\alpha}_{g,R}\Big) \wedge \bigwedge_{j\in MSG(TOP(k))} \widehat{\beta}_{j,R}$$

$$\widehat{\alpha}_{g,R} = \bigwedge_{\substack{k\in[r..r+|AOS(g)|-2]\\ n\in[1..R]}} ((\neg(OS_{k+1}[n]))\widetilde{\mathcal{U}}(OS_k[n])) \wedge \bigwedge_{\substack{OS_e\in AOS(g)\\ n\in[1..R]}} (\neg OS_e[n]\,\widetilde{\mathcal{U}}\,(OS_e[n] \wedge \bigcirc\Box\neg OS_e[n]))$$

$$\widehat{\beta}_{j,R} = \bigwedge_{n\in[1..R]} ((\neg RCV(j)[n])\,\widetilde{\mathcal{U}}\,(SND(j)[n]))$$

$$\widehat{\gamma}_{i,R} = \bigwedge_{OS\in TOS(CF[1]\uparrow_i)} (\neg OS\,\widetilde{\mathcal{U}}\,(\bigwedge_{OS_{pre}\in pre(CF[1]\uparrow_i)} (\diamondsuit OS_{pre})))$$
$$\wedge \bigwedge_{OS\in TOS(CF[R]\uparrow_i)} ((\bigwedge_{OS_{post}\in post(CF[R]\uparrow_i)} (\neg OS_{post}))\,\widetilde{\mathcal{U}}\,(\diamondsuit OS))$$

$$\kappa_{i,R} = \bigwedge_{n\in[1..R-1]} ((\bigwedge_{OS_q\in AOS(CF\uparrow_i)} (\neg OS_q[n+1]))\widetilde{\mathcal{U}}(\bigwedge_{OS_p\in AOS(CF\uparrow_i)} (\diamondsuit OS_p[n])))$$

Figure 4.16: LTL formula for fixed Loop

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 $maxint$.

4.2.5 Negation

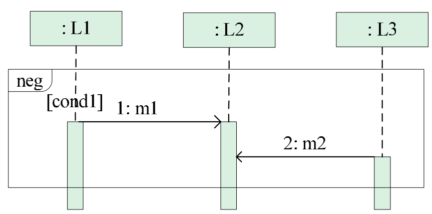


Figure 4.17: Example for Negative

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 4.17 [*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 invalid traces or the Operand is excluded.

4.2.6 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 Ψ_{assert}^{CF} in figure 4.19. θ^{CF} and γ_i^{CF} have their usual meanings. Function $\lambda_{(pre(CF\uparrow_i),AOS(CF\uparrow_i))}^{i,seq}$ represents that on Lifeline i, if all the OSs in the set of pre happen, no other OSs in Sequence Diagram seq are

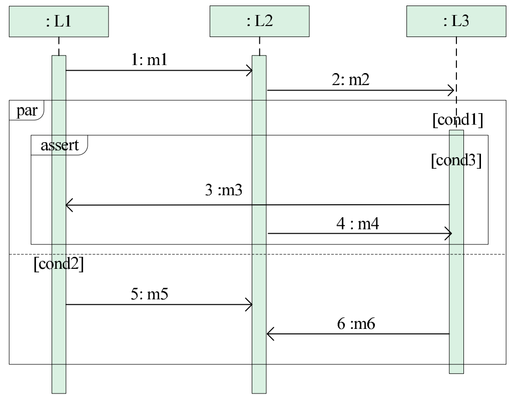


Figure 4.18: Example for Assertion

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 4.18), an Assertion is nested within a Parallel. The OSs within the CEU of the Assertion execute right after their preceding OSs finish execution. On Lifeline *L3*, after the execution of OS *r2*, OSs *s3* and *r4* must happen without being interleaved by OS *s6*.

$$\Psi^{CF}_{assert} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma^{CF}_i \wedge \bigwedge_{i \in LN(CF)} \lambda^{i,seq}_{(pre(CF\uparrow_i), AOS(CF\uparrow_i))}$$

$$\lambda^{i,seq}_{N_1,N_2} = \Box(\bigwedge_{OS_p \in N_1} (\diamondminus OS_p) \rightarrow ((\bigwedge_{OS_q \in (AOS(seq\uparrow_i) \setminus N_2)} (\neg OS_q)) \,\widetilde{\mathcal{U}}\, (\bigwedge_{OS_r \in N_2} (\diamondminus OS_r))))$$

Figure 4.19: LTL formula for Assertion

4.2.7 Weak Sequencing

The Weak Sequencing restricts the execution orders among its Operands along each Lifeline Figure 4.20 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 (see figure 4.21)..

Templates θ^{CF} and γ_i^{CF} have their usual meaning. γ_i^m specifies the execution orders between adjacent Operands, as well as enforcing the Weak Sequencing between the CF and its

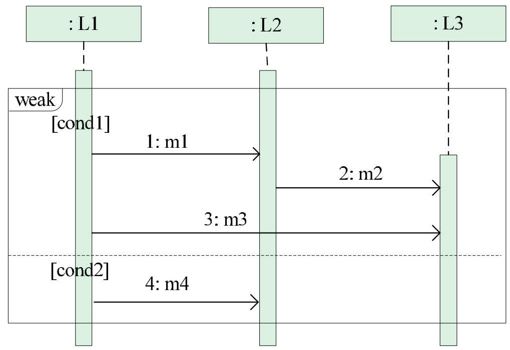


Figure 4.20: Example for Weak Sequencing

$$\Psi^{CF}_{weak} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF} \wedge \bigwedge_{i \in LN(CF)} \Big(\bigwedge_{m \in TOP(CF)} \gamma_i^{m} \Big)$$

Figure 4.21: LTL formula for Weak Sequencing

preceding/succeeding Interaction Fragments (γ_i^{CF}). (The LTL formula keeps γ_i^{CF} for clarity and consistency.)

4.2.8 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 together. (Weak Sequencing enforces the order among Operands on each Lifeline.) For example, OS *s4* will not execute until all OSs within the first Operand, including *s1*, *r1*, *s2*, *r2*, *s3*, and *r3* complete execution.

Figure 4.23 presents the semantics of Strict Sequencing. Template θ^{CF} has its usual meaning. The Strict Sequencing and its adjacent Interaction Fragments are connected using Weak Sequencing, which is expressed by template γ_i^{CF} as usual. Function χ_k asserts the order between each Operand k and its preceding Operand whose Constraint evaluates to *True*. Function $preEU(u)$

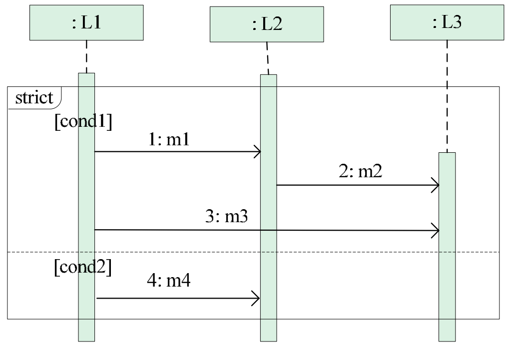


Figure 4.22: Example for Strict Sequencing

returns the set of OSs within EU v which happen right before EU u, *i.e.*, the Constraint of EU v evaluates to *True*. Function $NFTOP(CF)$ returns the set of Interaction Operands whose Constraints evaluate to *True* within *CF*, excluding the first one.

$$\Psi^{CF}_{strict} = \theta^{CF} \quad \wedge \quad \bigwedge_{i \in LN(CF)} \gamma_i^{CF} \quad \wedge \quad \bigwedge_{k \in NFTOP(CF)} \chi_k$$

$$\chi_k = (((\bigwedge_{OS \in AOS(k)} (\neg OS)))\ \widetilde{\mathcal{U}}\ (\bigwedge_{i \in LN(CF)} (\bigwedge_{OS_{pre} \in preEU(k\uparrow_i)} (\diamondminus OS_{pre}))))$$

Figure 4.23: LTL formula for Strict Sequencing

4.2.9 Coregion

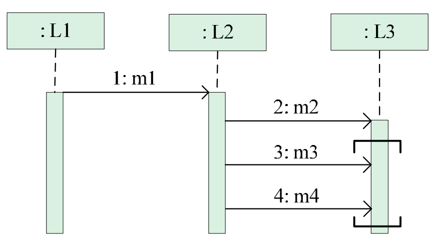


Figure 4.24: Example for Coregion

A Coregion is an area of a single Lifeline, which is semantically equivalent to a Parallel that the OSs are unordered. Figure 4.24 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 as a Parallel (see figure 4.25). Each OS within the Coregion is considered as an Operand of the Parallel, no order of OSs within a BEU needs to be defined. Template θ^{CF} is excluded because a Coregion does not contain any complete Messages. Complete messages are defined by the CF or Sequence Diagram which directly encloses them. γ_i^{CF} describes the Weak Sequencing between Coregion and its preceding/succeeding set of OSs. The LTL formula does not describe the Messages containing the OSs of the Coregion.

$$\boxed{\Psi_{coregion}^{CF} = \gamma_i^{CF}}$$

Figure 4.25: LTL formula for Coregion

4.3 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 4.26). 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 4.27). 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 [60]. For Ignore, the ignored message types are suppressed in the CF [60].

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 4.28). θ^{CF} and γ_i^{CF} have their usual meaning, which describe the semantics of Messages appearing in the Ignore. Each OS of the ignored Messages executes only once, which is enforced by $\widetilde{\alpha}_{ignoreOS(CF)}$. We introduce function $ignoreMsg(CF)$ to return the set of Messages of the ignored message types which occur

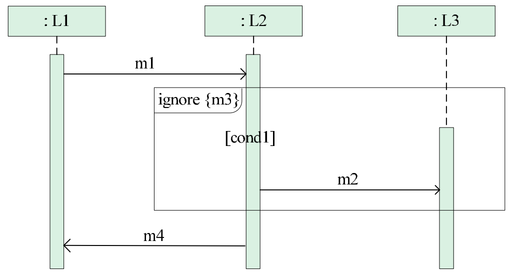


Figure 4.26: Example for Ignore

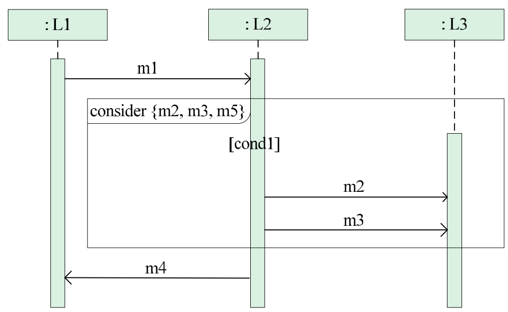


Figure 4.27: Example for Consider

in CF, which can be finite or infinite. Function $ignoreOS(CF)$ returns the set of OSs associated with Messages of ignored message types, which can also be finite or infinite. Formula β_k enforces that, for each ignored Message k, its sending OS must happen before its receiving OS. Formula $\gamma^{CF}_{i,ignoreOS(CF\uparrow_i)}$ extends γ^{CF}_i, which enforces any OS of the set of the ignored OSs can only happen within the CEU of the Ignore on each Lifeline, formally,

$$\gamma^{CF}_{i,\mathcal{S}} = \bigwedge_{OS \in \mathcal{S}} ((\neg OS \, \widetilde{\mathcal{U}} \, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\Diamond OS_{pre}))) \wedge ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \, \widetilde{\mathcal{U}} \, (\Diamond OS)))$$

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

$$\varepsilon_{seq,ignoreOS(CF)} = \Box((\widehat{\bigvee_{OS_p \in (AOS(seq) \cup ignoreOS(CF))}} OS_p) \vee (\bigwedge_{OS_p \in (AOS(seq) \cup ignoreOS(CF))} (\Diamond\!\!\!-OS_p)))$$

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

As the dual Operator of *ignore*, the semantics of a CF with Operator *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 4.29). Function $AllMsg(CF) \setminus considerMsg(CF)$ returns the Messages which are not considered but occur in CF, where $AllMsg(CF)$ returns all possible Messages, including Messages of considered types and Messages of ignored types. $considerMsg(CF)$ returns the Messages of considered types. Function $\Sigma \setminus considerOS(CF)$ returns all possible OSs within CF except the OSs of considered Messages, where Σ is the set of all possible OSs including considered OSs and ignored OSs, and $considerOS(CF)$ returns the set of OSs of considered Messages. In this way, the Sequence Diagram with Consider or Ignore no longer derive complete traces.

$$\Psi^{CF}_{ignore} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma^{CF}_i \wedge \widetilde{\alpha}_{ignoreOS(CF)} \wedge \bigwedge_{k \in ignoreMsg(CF)} \beta_k \wedge \bigwedge_{i \in LN(CF)} \gamma^{CF}_{i,ignoreOS(CF\uparrow_i)}$$

$$\widetilde{\alpha}_{\mathcal{S}} = \bigwedge_{OS_e \in \mathcal{S}} (\neg OS_e \,\widetilde{\mathcal{U}}\, (OS_e \wedge \bigcirc\Box\neg OS_e))$$

Figure 4.28: LTL formula for Ignore

$$\Psi_{consider}^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF} \wedge \widetilde{\alpha}_{\Sigma \setminus considerOS(CF)} \wedge \bigwedge_{k \in (AllMsg(CF) \setminus considerMsg(CF))} \beta_k \wedge \bigwedge_{i \in LN(CF)} \gamma_{i,(\Sigma_i \setminus considerOS(CF\uparrow_i))}^{CF}$$

Figure 4.29: LTL formula for Consider

4.4 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 [51]. In the following subsections, we discuss how to user our LTL framework to formalize the variations of Negative, Strict Sequencing, and Interaction Constraints.

4.4.1 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 *False*, the interpretation of the semantics of S may be varied, depending on the requirement of applications. Formula Ψ_{neg}^{S} instantiates the template Ψ_{neg}^{CF} (see subsection 4.2.5) with S, defining the traces of S, which can be invalid or inconclusive. For example, three traces defined by the Negative (see figure 4.17), [*s1, s2, r1, r2*], [*s2, s1, r1, r2*], and [*s1, r1, s2, r2*], can be interpreted as invalid, or inconclusive traces if $cond1$ evaluates to $False$.

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 $False$), the traces of R are also inconclusive traces. For Sequence Diagram R, its traces are defined by formula Π_R, which instantiates the template Π_{seq} (see figure 4.4). For non-Negative CF R, its traces are defined by formula Φ^R, which instantiates the template Φ^{CF} (see figure 4.5). For example, trace [*s1, s2, r2, r1, s3, r3*] in figure 4.30 is interpreted as an invalid or an inconclusive trace.

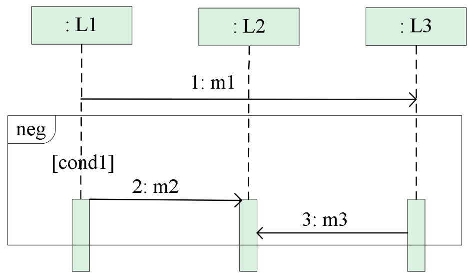


Figure 4.30: Example for variation of Negative Combined Fragment

For nested Negative CFs, *i.e.*, Negative CF R encloses Negative CF S, the traces of R are defined by Φ^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 $True$, 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 $False$, 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 4.31 shows an example of nested Negative CFs. All the traces [*s1, s2, r1, r2*], [*s2, s1, r1, r2*], and [*s1, r1, s2, r2*] of R can be valid, invalid, or inconclusive traces depending on the value of $cond1$ and $cond2$.

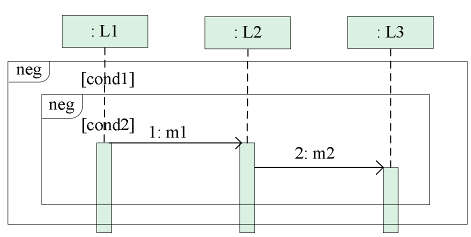


Figure 4.31: Example for nested Negative Combined Fragments

4.4.2 Variations of Strict Sequencing

Recall that 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 4.32). The only change we need to make is to replace γ_i^{CF} that enforces Weak Sequencing between the Strict Sequencing and its preceding/succeeding Interaction Fragments with ν^{CF}. Function ν^{CF} enforces the synchronization among multiple Lifelines when entering or leaving the Strict Sequencing, *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 ν^{CF} to replace function γ_i^{CF} in the LTL formula of the CF.

$$
\begin{aligned}
\Psi_{strict}^{CF'} &= \theta^{CF} \quad \wedge \bigwedge_{k \in NFTOP(CF)} \chi_k \quad \wedge \quad \nu^{CF} \\
\nu^{CF} &= ((\bigwedge_{i \in LN(CF)} (\bigwedge_{OS \in TOS(CF\uparrow_i)} (\neg OS))) \,\widetilde{\mathcal{U}}\, (\bigwedge_{i \in LN(CF)} (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre})))) \\
&\quad \wedge ((\bigwedge_{i \in LN(CF)} (\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post}))) \,\widetilde{\mathcal{U}}\, (\bigwedge_{i \in LN(CF)} (\bigwedge_{OS \in TOS(CF)} (\diamondminus OS))))
\end{aligned}
$$

Figure 4.32: LTL formula for variation of Strict Sequencing

4.4.3 Variations of Interaction Constraint

Recall that a CF consists of one or more Interaction Operands, each of which may contain an Interaction Constraint. An Interaction Constraint is located on a Lifeline with the first OS occurring within the Operand, *i.e.*, an Interaction Constraint is positioned above the first OS occurring within the Operand. For example, figure 2.1b contains a Parallel covering three Lifelines. In the first Operand *op1* of the Parallel, either OS *s2* or OS *s3* may be the first OS to execute. As the Interaction Constraint of *op1*, *cond1*, located on Lifeline *L2*, OS *s2* executes before OS *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 4.36, Interaction Constraint *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 *L2*. However, OS *s1* and OS *s2*, either of which may be the first one to execute within the Parallel, are located on *L1*. To avoid the contradiction, we assume 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 *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 4.37). Function *Init(m)* returns the first OS occurring within Operand *m*, which may return an empty set if the Interaction Constraint is located above a nested CF.

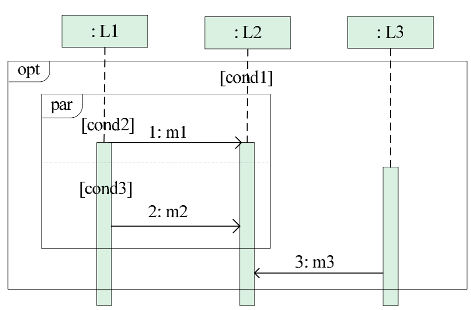


Figure 4.33: Example for CF with Interaction Constraints

Two different semantic interpretations of an Operand whose Interaction Constraint evaluates to $False$ are provided. 1. The traces expressed by the Operand are interpreted as invalid traces.

$$\mu^{CF} = \bigwedge_{m \in TOP(CF)} \Big(\bigwedge_{\substack{OS_p \in Init(m) \\ OS_q \in TOS(m)}} (\neg OS_q \,\widetilde{\mathcal{U}}\, OS_p) \Big)$$

Figure 4.34: LTL formula for Constraint of the first occurring OS

2. The Operand is excluded. Our LTL template chooses the second interpretation but also can be adapted to describe the first interpretation.

4.5 Other Control Constructs

4.5.1 General Ordering

General Ordering imposes order of two unorder 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.

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

4.5.2 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 *ref*. Formula Ψ^{CF}_{ref} represents the LTL representation of an Interaction Use. In Ψ^{CF}_{ref}, 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 γ^{CF}_i.

$$\Psi_{ref}^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF}$$

4.5.3 Discussion

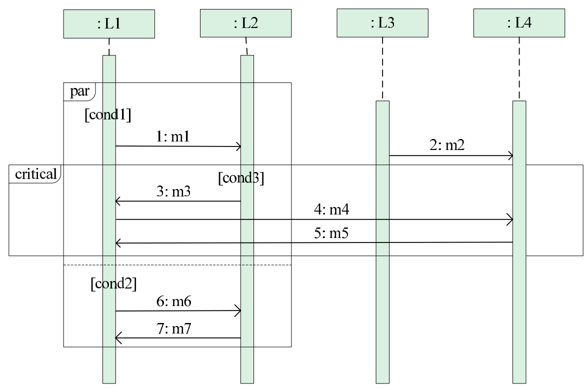


Figure 4.35: Example for Overlapped CFs

Our work does not address timed events, *i.e.*, the events cannot represent the occurrence of an absolute time. As the syntactic definition for OS, we do not handle the case that two OSs are overlapped on a Lifeline, *i.e.*, the relation between the set of OSs and the set of locations is one-to-one correspondence. The Messages are disallowed to cross the boundaries of CFs and their Operands [55]. 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, *i.e.*, the inner CF can not cover more Lifelines than the outer CF. The example in figure 4.35 has 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 is located on a Lifeline with the first OS occurring within the Operand, *i.e.*, an Interaction Constraint is positioned above the first OS occurring within the Operand. For example, figure 2.1b contains a Parallel covering three Lifelines. In the first Operand *op1* of the Parallel, either OS *s2* or OS *s3* may be the first OS to execute. As the Interaction Constraint of *op1*, *cond1*, located on Lifeline *L2*, OS *s2* executes before OS *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 4.36, Interaction Constraint *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 *L2*. However, OS *s1* and OS *s2*, either of which may be the first one to execute within the Parallel, are located on *L1*. To avoid the contradiction, we assume 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 *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 4.37). Function *Init(m)* returns the first OS occurring within Operand *m*, which may return an empty set if the Interaction Constraint is located above a nested CF.

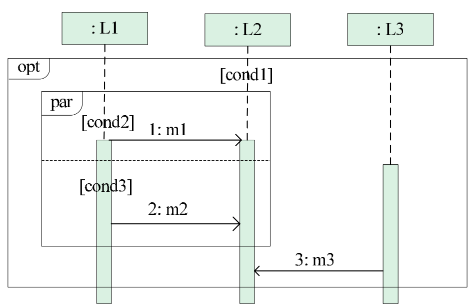


Figure 4.36: Example for CF with Interaction Constraints

$$\mu^{CF} = \bigwedge_{m \in TOP(CF)} \Big(\bigwedge_{\substack{OS_p \in Init(m) \\ OS_q \in TOS(m)}} (\neg OS_q \, \widetilde{\mathcal{U}} \, OS_p) \Big)$$

Figure 4.37: LTL formula for Constraint of the first occurring OS

4.6 Proof for LTL Template of Sequence Diagram with Combined Fragments

We wish to prove that the NuSMV model for a Sequence Diagram with CFs capture the semantics of the Sequence Diagram. Recall the semantic rules general to all CFs have been presented in section 2.2, and the semantics of each CF Operator is shown in section 2.3. The LTL template for Sequence Diagram with CFs, Π_{seq}, is shown in figure 4.4 (see section 4).

$$\boxed{\widetilde{\Pi}_{seq} = (\bigwedge_{\substack{i \in LN(seq) \\ g \in TBEU(seq\uparrow_i)}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq)} \rho_j) \wedge (\bigwedge_{j \in MSG(seq)} \beta_j) \wedge (\bigwedge_{CF \in nested(seq)} \Phi^{CF}) \wedge \varepsilon_{seq}}$$

Figure 4.38: Rewriting LTL templates for Sequence Diagram with Combined Fragments

$$\begin{aligned}
\Pi_{seq} &= \bigwedge_{i \in LN(seq)} (\bigwedge_{g \in TBEU(seq\uparrow_i)} \alpha_g) \wedge \bigwedge_{j \in MSG(seq)} \beta_j \wedge \bigwedge_{CF \in nested(seq)} \Phi^{CF} \wedge \varepsilon_{seq} \\
&= (\bigwedge_{\substack{i \in LN(seq) \\ g \in TBEU(seq\uparrow_i)}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq)} \rho_j) \wedge (\bigwedge_{j \in MSG(seq)} \beta_j) \wedge (\bigwedge_{CF \in nested(seq)} \Phi^{CF}) \wedge \varepsilon_{seq}
\end{aligned}$$

Figure 4.39: Rewriting Π_{seq} into $\widetilde{\Pi}_{seq}$

We can write LTL template Π_{seq} into $\widetilde{\Pi}_{seq}$ (see figure 4.38) by replacing the sub-formula $\bigwedge_{\substack{i \in LN(seq) \\ g \in TBEU(seq\uparrow_i)}} \alpha_g$ using sub-formulas $\bigwedge_{\substack{i \in LN(seq) \\ g \in TBEU(seq\uparrow_i)}} \tilde{\alpha}_g$ and $\bigwedge_{j \in MSG(seq)} \rho_j$. The procedure (see figure 4.39) follows the one of rewriting LTL template Π_{seq}^{Basic}. We can rewrite sub-formula θ^{CF} into $\tilde{\theta}^{CF}$ (see figure 4.40) to describe the semantics of CF's Operands whose Constraints evaluate to *True* (see figure 4.41). In sub-formula θ^{CF}, function $TBEU(CF\uparrow_i)$ returns the set of BEUs, whose Constraints evaluate to *True*, directly enclosed in the CEU of CF on Lifeline i. It is equivalent to the set of BEUs directly enclosed in the EUs, which are obtained by projecting CF's Operands whose Constraints evaluate to *True* onto Lifeline i, *i.e.*, $TBEU(CF\uparrow_i) = \{beu | beu \in ABEU(op\uparrow_i) \wedge op \in TOP(CF)\}$ (see line 1). Sub-formula $\bigwedge_{\substack{i \in LN(CF) \\ g \in ABEU(op\uparrow_i)}} \alpha_g$ is rewritten as the one of rewriting Π_{seq}^{basic} (see line 4). We also rewrite sub-formula γ_i^{CF} into $\tilde{\gamma}_i^{CF}$ (see figure 4.40) to enforce the sequential execution Lifeline i (see figure 4.42). In sub-formula γ_i^{CF}, function $TOS(CF\uparrow_i)$ returns the set of OSs of the BEUs, whose Constraints evaluate to *True*, directly enclosed in the CEU of CF on Lifeline i. It is equivalent to the set of OSs directly enclosed in the Operands whose Constraints evaluate to *True* on Lifeline i, *i.e.*, $TOS(CF\uparrow_i) = \{os|os \in AOS(ABEU(op\uparrow_i)) \wedge op \in TOP(CF)\}$.

$$
\boxed{
\begin{aligned}
\Psi^{CF} &= \tilde{\theta}^{CF} \wedge \bigwedge_{i \in LN(CF)} \tilde{\gamma}_i^{CF} \wedge \varrho^{CF} \\
\tilde{\theta}^{CF} &= \bigwedge_{op \in TOP(CF)} \Big(\Big(\bigwedge_{\substack{i \in LN(CF)\\ g \in ABEU(op\uparrow_i)}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j \in MSG(op)} \rho_j\Big) \wedge \Big(\bigwedge_{j \in MSG(op)} \beta_j\Big)\Big) \\
\tilde{\gamma}_i^{CF} &= \bigwedge_{op \in TOP(CF)} \Big(\bigwedge_{\substack{beu \in ABEU(op\uparrow_i)\\ OS \in AOS(beu)}} \Big(\Big(\neg OS\, \widetilde{\mathcal{U}}\, \Big(\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre})\Big)\Big) \wedge \Big(\Big(\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})\Big)\, \widetilde{\mathcal{U}}\, (\diamondminus OS)\Big)\Big)\Big)
\end{aligned}
}
$$

Figure 4.40: Rewriting LTL template for OSs directly enclosed in Combined Fragment

$$
\begin{aligned}
\theta^{CF} &= \Big(\bigwedge_{i \in LN(CF)} \Big(\bigwedge_{g \in TBEU(CF\uparrow_i)} \alpha_g\Big)\Big) \wedge \Big(\bigwedge_{j \in MSG(TOP(CF))} \beta_j\Big) \\
&= \Big(\bigwedge_{i \in LN(CF)} \Big(\bigwedge_{op \in TOP(CF)} \Big(\bigwedge_{g \in ABEU(op\uparrow_i)} \alpha_g\Big)\Big)\Big) \wedge \Big(\bigwedge_{op \in TOP(CF)} \Big(\bigwedge_{j \in MSG(op)} \beta_j\Big)\Big) && (1) \\
&= \Big(\bigwedge_{op \in TOP(CF)} \Big(\bigwedge_{i \in LN(CF)} \Big(\bigwedge_{g \in ABEU(op\uparrow_i)} \alpha_g\Big)\Big)\Big) \wedge \Big(\bigwedge_{op \in TOP(CF)} \Big(\bigwedge_{j \in MSG(op)} \beta_j\Big)\Big) && (2) \\
&= \bigwedge_{op \in TOP(CF)} \Big(\Big(\bigwedge_{\substack{i \in LN(CF)\\ g \in ABEU(op\uparrow_i)}} \alpha_g\Big) \wedge \Big(\bigwedge_{j \in MSG(op)} \beta_j\Big)\Big) && (3) \\
&= \bigwedge_{op \in TOP(CF)} \Big(\Big(\bigwedge_{\substack{i \in LN(CF)\\ g \in ABEU(op\uparrow_i)}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j \in MSG(op)} \rho_j\Big) \wedge \Big(\bigwedge_{j \in MSG(op)} \beta_j\Big)\Big) && (4) \\
&= \tilde{\theta}^{CF}
\end{aligned}
$$

Figure 4.41: Rewriting θ^{CF} into $\tilde{\theta}^{CF}$

Lemma 4.10. *A given Sequence Diagram with CFs, seq, directly contains h Message. In the CFs, p Messages are enclosed in Operands whose Interaction Constraints evaluate to True,* i.e.*, if a Message is enclosed in multiple nested Operands, all the Interaction Constraints of the Operands evaluate to True. For other q Messages within the CFs, each Message is enclosed in one Operand*

$$
\begin{aligned}
\gamma_i^{CF} &= \bigwedge_{OS \in TOS(CF\uparrow_i)} ((\neg OS\, \widetilde{\mathcal{U}}\, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre}))) \wedge ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post}))\, \widetilde{\mathcal{U}}\, (\diamondminus OS))) \\
&= \bigwedge_{op \in TOP(CF)} (\bigwedge_{\substack{beu \in ABEU(op\uparrow_i) \\ OS \in AOS(beu)}} ((\neg OS\, \widetilde{\mathcal{U}}\, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondminus OS_{pre}))) \wedge ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post}))\, \widetilde{\mathcal{U}}\, (\diamondminus OS)))) \\
&= \tilde{\gamma}_i^{CF}
\end{aligned}
$$

Figure 4.42: Rewriting γ_i^{CF} into $\tilde{\gamma}_i^{CF}$

or multiple nested Operands, where at least one Operand's Interaction Constraint evaluate to False. If $\sigma \in (\Sigma_{LTL}^{seq})^\omega$, then σ must have the form, i.e., *$\sigma = \sigma_{[1..2h+2p]} \cdot \tau^\omega$, where $\sigma_{[1..2h+2p]}$ contains no τ.*

Proof. If $\sigma \models \widetilde{\Pi}_{seq}$, then $\sigma \models \bigwedge_{j \in MSG(seq)} \rho_j$ and $\sigma \models \bigwedge_{op \in TOP(CF)} (\bigwedge_{j \in MSG(op)} \rho_j)$. From subformula $\bigwedge_{j \in MSG(seq)} \rho_j$, we can infer that each OS of the Messages directly enclosed in seq can execute once and only once in σ. For each CF, we can infer from $\bigwedge_{op \in TOP(CF)} (\bigwedge_{j \in MSG(op)} \rho_j)$ that each OS of the Messages directly enclosed in CF's Operands whose Constraints evaluate to *True* can execute once and only once. Similarly, if $\sigma \models \Pi_{seq}$, we can deduce that $\sigma \models \varepsilon_{seq}$. It specifies that only one enabled OS (*i.e.*, the OS is not enclosed in an Operand whose Constraint evaluates to *False*) can execute at a time, and σ should execute uninterrupted until all the enabled OSs have taken place. seq directly contains h Messages with $2h$ OSs. In the CFs within seq, the Operands whose Interaction Constraints evaluate to *True* contain p Messages with $2p$ OSs. Therefore, σ should have the form, $\sigma = \sigma_{[1..2h+2p]} \cdot \tau^\omega$, where $\sigma_{[1..2h+2p]}$ contains no τ. $\square$

A given Sequence Diagram, seq_r, directly contains k Lifelines, h Messages and r CFs, which contain $p + q$ Messages. Each CF does not contain other CFs. For the Messages within the CFs, p Messages are enclosed in Operands whose Interaction Constraints evaluate to *True*, while q Message are enclosed in Operands whose Interaction Constraints evaluate to *False*.

We wish to prove that $\forall \upsilon . \upsilon \in (\Sigma_{sem}^{seq_r})^*, \upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_r}$, *i.e.*, $\upsilon \cdot \tau^\omega \in (\Sigma_{LTL}^{seq_r})^\omega$. The semantic rules of seq_r define that each OS which is directly enclosed in seq_r or an Operand whose Constraint evaluates to *True*, occurs once and only once. Thus, $\forall \upsilon . \upsilon \in (\Sigma_{sem}^{seq_r})^*, |\upsilon| = 2h + 2p$. From lemma 4.10, we learn that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq_r})^\omega, \sigma = \sigma_{[1..2h+2p]} \cdot \tau^\omega$, where $\sigma_{[1..2h+2p]}$ contains no τ. $\sigma_{[1..2h+2p]} \in PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega)$. If $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_r})^*, \upsilon \cdot \tau^\omega \in (\Sigma_{LTL}^{seq_r})^\omega$, we can infer that, $\upsilon \in PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega)$, *i.e.*, $(\Sigma_{sem}^{seq_r})^* \subseteq PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega)$.

We also wish to prove that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq_r})^\omega, \sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_r})^*$, *i.e.*, $PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega) \subseteq (\Sigma_{sem}^{seq_r})^*$.

Theorem 4.11. *$(\Sigma_{sem}^{seq_r})^*$ and $PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega)$ are equal.*

We provide the proof of theorem 4.11 in appendix B.2.

We consider the Sequence Diagram with nested CFs. A given Sequence Diagram, seq_{nested}, directly contains k Lifelines, h Messages and r CFs, which contain $p+q$ Messages. Each CF may contain other CFs. We use layer to define the location of the nested CFs. The Sequence Diagram's layer is 0, while the layer of a CF directly enclosed in the Sequence Diagram is 1. If CF cf_m's layer is m, then the layer of the CFs directly enclosed in cf_m is $m+1$. We assume the maximum layer of CF within seq_{nested} is l. For the Messages within the CFs, p Messages are enclosed in Operands whose Interaction Constraints evaluate to *True*, *i.e.*, if a Message is enclosed in multiple nested Operands, all the Interaction Constraints of the Operands evaluate to *True*. For other q Messages within the CFs, each Message is enclosed in one Operand or multiple nested Operands, where at least one Operand's Interaction Constraint evaluate to *False*. We wish to prove that $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_{nested}})^*, \upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{nested}}$, *i.e.*, $\upsilon \cdot \tau^\omega \in (\Sigma_{LTL}^{seq_{nested}})^\omega$. The semantic rules of seq_{nested} define that each OS which is directly enclosed in seq_r or Operands whose Constraints evaluate to *True*, occurs once and only once. Thus, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_{nested}})^*, |\upsilon| = 2h+2p$. From lemma 4.10, we learn that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq_{nested}})^\omega, \sigma = \sigma_{[1..2h+2p]} \cdot \tau^\omega$, where $\sigma_{[1..2h+2p]}$ contains no τ. $\sigma_{[1..2h+2p]} \in PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega)$. If $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_{nested}})^*, \upsilon \cdot \tau^\omega \in (\Sigma_{LTL}^{seq_{nested}})^\omega$, we can infer that, $\upsilon \in PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega)$, *i.e.*, $(\Sigma_{sem}^{seq_{nested}})^* \subseteq PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega)$.

We also wish to prove that $\forall \sigma.\sigma \in (\Sigma_{LTL}^{seq_{nested}})^\omega, \sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_{nested}})^*$, *i.e.*, $PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega) \subseteq (\Sigma_{sem}^{seq_{nested}})^*$.

Theorem 4.12. *$(\Sigma_{sem}^{seq_{nested}})^*$ and $PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega)$ are equal.*

We provide the proof of theorem 4.12 in appendix B.2.

Chapter 5: EXPRESSING SAFETY PROPERTIES USING SEQUENCE DIAGRAMS

Practitioners tend to construct multiple Sequence Diagrams to capture the requirements and design of a system. A Sequence Diagram may present a possible execution, describing how the environment and system interact with each other, or specifies core requirements or a desired property. 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 a model Sequence Diagram. In this section we present how to generate safety properties as LTL formulas from Sequence Diagrams with Negative and Assertion respectively.

5.1 Safety Property with Negative Combined Fragment

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 forbidden traces exist. Two types of safety properties are provided: the strong safety property and the weak safety property. A system is strong safe with respect to a Negative if any run of the system is not compatibly induced by a trace which contains the OSs of the Negative and the OSs are ordered as an invalid trace. The strong safety properties focus on the order of OSs of invalid traces, which can be specified as an LTL template Ω_{seq}^{SNCF},

$$\begin{aligned}
\Omega_{seq}^{SNCF} =& \neg(\Phi^{CF} \wedge \varepsilon_{seq}^{part}) \\
\varepsilon_{seq}^{part} =& \Box((\widehat{\bigvee_{OS_m \in AOS(seq)}} OS_m) \vee (\bigwedge_{OS_m \in AOS(seq)} (\neg OS_m)))
\end{aligned}$$

where formula Φ^{CF} asserts the order of OSs enclosed in the Negative, and ε_{seq}^{part} enforces the interleaving semantics of partial traces, *i.e.*, at most one OS of a Sequence Diagram seq can execute at once; other OSs may occur and interleave the partial traces.

If there is no run of the system, which is compatibly induced by a trace containing an invalid trace of the Negative as a sub-trace, we consider the system is weak safe with respect to the Negative. If a system is weak safe but not strong safe with respect to the Negative, we consider the traces which violate strong safety properties as false positive traces. We define a temporal logic template, Ω_{seq}^{WNCF}, to characterize the weak safety property of Sequence Diagram seq with respect to a Negative. Formally,

$$\Omega_{seq}^{WNCF} = \neg(\Phi^{CF} \wedge \varepsilon_{seq}^{part} \wedge \delta_{(AOS(NCF),(AOS(seq) \setminus AOS(NCF)))})$$

in which formula $\delta_{(AOS(NCF),(AOS(seq) \setminus AOS(NCF)))}$ asserts that the traces enclosed in the Negative are not interleaved by other OSs in Sequence Diagram seq. Formula Ω_{seq}^{WNCF} asserts that Sequence Diagram seq is not weak safe if 1. A trace of seq contains the OSs of an invalid trace and these OSs are ordered as the invalid trace (first conjunct). 2. these OSs are not interleaved by other OSs of seq (second conjunct).

Figure 5.1 shows an example of a Negative, which we want to verify against the Sequence Diagram, seq, shown in figure 2.1a. Our techniques define a strong safety property Ω_{seq}^{SNCF} and a weak safety property Ω_{seq}^{WNCF} with respect to the example Negative, which can verify the model translated from seq. The strong safety property is violated because seq contains a trace [*s1, r1, s2, r2, s3, r3*], which orders OSs *s1, r1, s3* and *r3* as the Negative. The weak safe property is satisfied in that invalid traces [*s1, r1, s3, r3*] and [*s1, s3, r1, r3*] are not shown as sub-traces in the example Sequence Diagram.

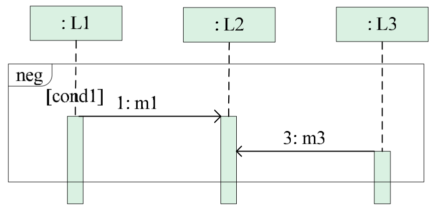


Figure 5.1: Example for Negative representing safety property

5.2 Safety Property with Assertion Combined Fragment

We define that a collection of Sequence Diagrams is safe 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 Ω_{seq}^{ASCF} in figure 5.3 represents the safety property of seq with respect to an Assertion. On Lifeline i, two conditions should be satisfied if all the OSs in the set of 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 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 $\Phi^{CF_j}\uparrow_i$, which is the restriction of Φ^{CF_j} on Lifeline *i*. Generally, $\Phi^{CF_j}\uparrow_i$ is a conjunction of sub-formulas α, γ, and additional sub-formulas (optional and various for different CFs) on Lifeline *i*. To obtain $\Phi^{CF_j}\uparrow_i$, we need to project the syntactic constructs of CF_j on Lifeline *i*, and then keep the sub-formulas of Φ^{CF_j} which are related to these constructs.

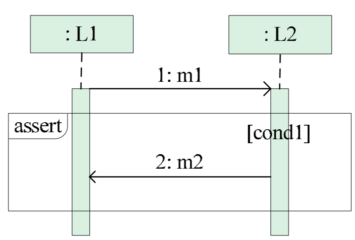


Figure 5.2: Example for Assertion representing safety property

Based on this safety definition, we can verify if a Sequence Diagram, seq, satisfies the safety constraints set by another Sequence Diagram with an Assertion. For example, we can verify

$$\Omega_{seq}^{ASCF} = \bigwedge_{i \in LN(CF)} (\Box(\bigwedge_{OS_p \in pre(CF\uparrow_i)} (\diamondminus OS_p) \rightarrow (((\bigwedge_{OS_q \in (AOS(seq\uparrow_i) \backslash AOS(CF\uparrow_i))} (\neg OS_q)) \\ \widetilde{\mathcal{U}} (\bigwedge_{OS_r \in AOS(CF\uparrow_i)} (\diamondminus OS_r))) \wedge (\bigwedge_{g \in TBEU(CF\uparrow_i)} \alpha_g) \wedge \bigwedge_{CF_j \in nested(CF)} (\Phi^{CF_j} \uparrow_i)))) \wedge \varepsilon_{seq}^{part}$$

Figure 5.3: Safety property for Assertion

Ω_{seq}^{ASCF} for the Sequence Diagram in figure 5.2 against the model for seq, in figure 2.1a. The safety property is violated, and a counterexample trace [*s1, r1, s5, r5, s2, s3, r3, s4, r4, r2, s6, r6*] is provided, where mandatory trace [*s1, r1, s2, r2*] is not always strictly included as a sub-trace.

5.3 Deadlock Property with Synchronous Messages

The deadlock-free property can be verified in a Sequence Diagram with synchronous Messages, where each synchronous Message must have an explicit reply Message (see example in figure 5.4). Deadlock can occur if multiple Lifelines are blocked, waiting on each other for a reply.

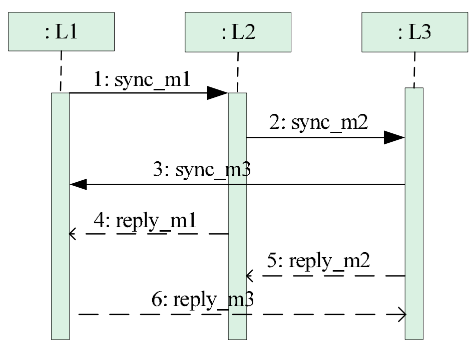


Figure 5.4: Example for deadlock in basic Sequence Diagram with synchronous Messages

Figure 5.5 represents the LTL formula of a basic Sequence Diagram with synchronous Messages, which conjuncts a constraint ξ_{seq}^{sync} with the LTL formula of a basic Sequence Diagram with asynchronous Messages. ξ_{seq}^{sync} describes 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 $typeOS(OS_p)$ returns that OS_p is a sending OS or a receiving OS, and $Reply(OS_p)$ returns the reply Message of a synchronous Message containing OS_p. In the example of figure 5.4, all of the Lifelines eventually deadlock since they all send Messages and are all awaiting replies. The LTL formula is not satisfied when verifying against the NuSMV module, which will be discussed in section 6.3.1.

$$\boxed{\begin{aligned}\Pi_{seq}^{sync} =& \Pi_{seq} \wedge \xi_{seq}^{sync}\\ \xi_{seq}^{sync} =& \Box \bigwedge_{i \in LN(seq)} \Big(\bigwedge_{\substack{OS_p \in AOS(seq\uparrow_i)\\ typeOS(OS_p)=send}} (OS_p \rightarrow ((\bigwedge_{\substack{OS_q \in AOS(seq\uparrow_i)\\ OS_p \neq OS_q}} (\neg OS_q))\, \mathcal{U}\, RCV(Reply(OS_p)))))\end{aligned}}$$

Figure 5.5: LTL formula for Sequence Diagram with synchronous Messages

5.4 Ignore and Consider within Properties

A property Sequence Diagram may consist of an Ignore or Consider CF. The Messages that are ignored in such a CF may interleave not only the subtraces of the CF (as we define in section 4), but also interleave the (partial) property trace. We need to define an LTL formula to address this issue.

$$\Psi_{ignore}^{CF} = \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF}$$

Figure 5.6: LTL formula for Ignore in property

$$\begin{aligned}\Psi_{consider}^{CF} =& \theta^{CF} \wedge \bigwedge_{i \in LN(CF)} \gamma_i^{CF} \wedge \zeta_{considerOS(CF) \setminus AOS(CF)}\\ \zeta_{\mathcal{S}} =& \Box(\bigwedge_{OS_n \in \mathcal{S}} (\neg OS_n))\end{aligned}$$

Figure 5.7: LTL formula for Consider in property

The LTL formula of Ignore within properties constrains the semantics of Messages appearing in the Ignore with formulas θ^{CF} and γ_i^{CF} (see figure 5.6). For Consider, the OSs of considered Messages which do not appear in the Consider can not occur to interleave the partial subtraces of the CF, which is captured by formula $\zeta_{considerOS(CF)\setminus AOS(CF)}$ (see figure 5.7). Function $considerOS(CF) \setminus AOS(CF)$ returns the OSs of the considered Message which do not appear in CF, where $considerOS(CF)$ returns the set of Messages of the considered message types.

Chapter 6: MAPPING SEQUENCE DIAGRAM TO NUSMV MODEL

In the previous sections, we presented a formal framework to formalize 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 verify properties specified using Negative and Assertion CFs.

In practice, software engineers often construct a collection of Sequence Diagrams that complement each other for specifying system requirements. In many cases, some Sequence Diagrams for a single system are intended to express valid traces, while others are to express that certain traces 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, the analytical power of NuSMV can be leveraged to determine whether the collection of Sequence Diagrams is safe (*i.e.*, the set of "intended" valid traces and the set of invalid traces are disjoint) and consistent (*i.e.*, the set of valid traces is a subset of the mandatory traces).

This section first provides 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 use examples to 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.

6.1 NuSMV Overview

NuSMV is a model checking tool, which exhaustively explores all execution traces 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 trace. 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 **init** statement of the form *init(x) := EXP*, which defines the value or set of values x can assume initially. Transitions are represented by using the **next** statements of the form *next(x) := EXP*, which defines the value or set of values that 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 *x := EXP* and they are replaced by *EXP* in each state. The system's invariant is represented with the **INVAR** statement, which is a boolean expression satisfied by each state.

6.2 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 formal 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, *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 *False* (note that a CEU module does not contain OS variables). Once an OS occurs, its value is set to *True* and then to *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, *state*, in each Lifeline or EU module. *state* assumes an enumeration element to express that respective OSs have taken place and an initial value, *sinit*, to express that no OSs have occurred yet. A CEU module contains one boolean variable, *cond*, for each of its EUs to represent the Interaction Constraint of the EU.

To express the interleaving semantics among Lifelines, we introduce an **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 *chosen* is used for each Lifeline to restrict that: (1) a Lifeline is chosen only if it is enabled, *i.e.*, there is an OS that is ready to take place on the Lifeline, represented by the derived variable *enabled*; (2) either only one Lifeline can be chosen to execute an OS in each step if Lifelines are enabled (*i.e.*, before all OSs on the Lifelines have occurred); or no Lifeline can be chosen when all Lifelines are not enabled and all *chosen* variables remain *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 *chosen* for each Lifeline is passed to the CEU and EU modules on that Lifeline as a parameter.

6.3 Basic Sequence Diagram

In this section, we provide a mapping strategy, and prove that it represents the semantics of a basic Sequence Diagram.

6.3.1 Basic Sequence Diagram with Asynchronous Messages

In this subsection, we illustrate our mapping strategy with an example basic Sequence Diagram as shown in figure 2.1a. Figure 6.1 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 *l_L1*, *l_L2*, and *l_L3* in the main module. We show the implementation of module *L2* here. Module *L2* takes modules *L1, L3* as parameters. Three OSs on Lifeline *L2* are defined as boolean variables *OS_r1*, *OS_r2*, and *OS_r3* in the VAR section. We define each OS as *OS_sx* or *OS_rx*, where *s* and *r* denote they are sending or receiving OSs, and *x* is the corresponding Message name. The enumerated variable *state* has four values, including a initial value *sinit* and three values to record the execution of the three respective OSs. A derived variable *enabled* for each OS represents the enabling condition of the OS by using the variable *state* in the DEFINE section. For instance, *r3_enabled* for OS *OS_r3* is *True* if and only if the sending OS of Message *m3* and the preceding OS, *OS_r2*, on Lifeline $L2$ have taken place, *i.e.*, *state* on Lifeline $L2$ sets to *r2* and *state* on Lifeline $L3$ sets to *s3*. The Lifeline *L2* can be enabled if and only if one of *r1, r2*, and *r3* is enabled. The variable *flag_final* checks whether the last OS *r3* on *L2* takes place (*i.e.*, *state* sets to *r3*. If so, all OSs in module *L2* have occurred. The ASSIGN section defines the transition relation of module *L2*. For example, *OS_r3* is set to *False* initially. When it is chosen and enabled, it is set to *True*. It is set to *False* in the subsequent states to represent that an OS can execute exactly once. Variable *state* is set to $r1$ in the same state in which OS_r1 occurs.

```
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.enabled & !l_L2.enabled & !l_L3.enabled)))



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;
  flag_final := state = r3;
 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;
```

Figure 6.1: Basic Sequence Diagram with asynchronous Messages to NuSMV

6.3.2 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 blocks until a reply Message is received. We introduce a boolean variable, *isBlock*, for each Lifeline to capture this semantic aspect. All OSs on a Lifeline include *isBlock* as part of their enabling conditions, thus preventing the OSs from occurring while *isBlock* is *True*.

Figure 6.2 represents the NuSMV description of a Sequence Diagram with synchronous Messages (see figure 5.4), containing a module for Lifeline *L1*. Each OS name is prefixed with either *sync* for a synchronous Message, or *reply* for a reply Message. Each OS has an enabling condition *!isblock* indicating that the OS can not be enabled when the Lifeline is blocked. *isBlock* initializes to $False$ and is set to $True$ when the sending OS *sync_s1* executes. It is set to $False$ when the OS *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 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;
  ...
```

Figure 6.2: Basic Sequence Diagram with synchronous Messages to NuSMV

6.3.3 Proof for NuSMV Model of Basic Sequence Diagram

We wish to prove that the NuSMV model for basic Sequence Diagram capture the semantics of basic Sequence Diagram. Recall the semantic aspects of basic Sequence Diagrams have been represented in section 2.1. Section 6.3.1 describes the mapping strategy for a basic Sequence Diagram, where an example of NuSMV model for basic Sequence Diagram is shown in figure 6.1.

We can generate all possible execution traces from a NuSMV model. Each execution trace consists of a sequence of states, each of which is an assignment of variable values. In the initial state, all the variables for OSs are initialized. In each following state, the value of the executing OS is changed, triggering the transition from the previous state to the current state. The value of $state$ of the Lifeline where the executing OS locates on is also changed to record the execution. Thus, an execution trace can be considered as a sequence of OSs. Each execution trace is infinite by stuttering at the final state if there is no more state change (*i.e.*, no more OS executes). We wish to prove that the finite prefix (without stuttering) of each trace generated from a NuSMV model represents a trace derived from the corresponding Sequence Diagram.

For a given basic Sequence Diagram, seq, with j Messages and $2j$ event occurrences, $\Sigma_{sem}^{seq} \subseteq \Sigma$ is the set of event occurrences of seq. The set of valid traces, $(\Sigma_{sem}^{seq})^*$, contains finite traces derived from seq based on the semantics of Sequence Diagrams. For the NuSMV model of seq, M_{seq}, Σ_{NuSMV}^{seq} is the set of event occurrences of M_{seq}, where $\Sigma_{NuSMV}^{seq} = \Sigma_{sem}^{seq} \cup \{\tau\}$. τ is an invisible event occurrence which does not occur in seq, *i.e.*, $\tau \in (\Sigma \setminus \Sigma_{sem}^{seq})$. $(\Sigma_{NuSMV}^{seq})^\omega$ represents all infinite traces that can be generated from M_{seq}.

Lemma 6.13. *For a given Sequence Diagram, seq, with j Messages, if $\sigma \in (\Sigma_{NuSMV}^{seq})^\omega$, then σ must have the form, $\sigma = \sigma_{[1..2j]} \cdot \tau^\omega$, where $\sigma_{[1..2j]}$ contains no τ.*

Proof. seq contains j Messages with $2j$ OSs. In the NuSMV model for seq, M_{seq}, the INVAR statement asserts that one enabled OS on one Lifeline can take place in each step until no more OSs are enabled. Therefore, no τ occurs between two OSs in σ. The variables of OSs in the

Lifeline modules define that each OS can execute once and only once. Thus, we can infer that σ have the form, $\sigma = \sigma_{[1..2j]} \cdot \tau^{\omega}$, where $\sigma_{[1..2j]}$ does not contain τ. $\square$

We wish to prove that for a given Sequence Diagram, seq, with j Messages, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^{*}, \upsilon \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq})^{\omega}$. The semantic rule of seq defines that each OS occurs once and only once. Thus, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^{*}, |\upsilon| = 2j$. From lemma 6.13, we learn that $\forall \sigma.\sigma \in (\Sigma_{NuSMV}^{seq})^{\omega}, \sigma = \sigma_{[1..2j]} \cdot \tau^{\omega}$, where $\sigma_{[1..2j]}$ contains no τ. $\sigma_{[1..2j]} \in PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega})$. If $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq})^{*}, \upsilon \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq})^{\omega}$, we can infer that, $\upsilon \in PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega})$, *i.e.*, $(\Sigma_{sem}^{seq})^{*} \subseteq PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega})$.

We also wish to prove that $\forall \sigma.\sigma \in (\Sigma_{NuSMV}^{seq})^{\omega}, \sigma_{[1..2j]} \in (\Sigma_{sem}^{seq})^{*}$, *i.e.*, $PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega}) \subseteq (\Sigma_{sem}^{seq})^{*}$.

Theorem 6.14. *For a given Sequence Diagram, seq, with j Messages, $(\Sigma_{sem}^{seq})^{*}$ and $PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega})$ are equal.*

We provide the proof of theorem 6.14 in appendix B.3.

6.4 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 *cond*. We do not control the value of *cond* until the Operand is ready to enter, representing the fact that a condition may change during the execution of the Sequence Diagram. If *cond* evaluates to *True*, the Operand is entered, otherwise, the Operand is skipped. Afterwards, the value of *cond* stays the same. While there is no Constraint in an Operand, *cond* is defined as constant *True*. Thus, the NuSMV implementation of Interaction Constraints is consistent with the LTL semantics of the Constraints. An extra variable *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 *False*) by enumeration element *0*, and “start” (Constraint evaluates to *True*) by enumeration element *1*. *cond* is evaluated when the Operand is ready to be entered, *i.e.*, *op_eva* evaluates to either *0* or *1*. Both *cond* and *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 *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.

6.4.1 Concurrency

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 *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 *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 6.4.2.

6.4.2 Atomic execution

A Critical Region has a sole Operand with each CEU module having a single EU submodule. We use a boolean variable, *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 *isCritical* is initialized to *False* in each EU module of the Critical Region's Operand. It is set to *True* if the EU starts to execute OSs and stays *True* until the EU finishes execution. Once the EU completes, *isCritical* is set to *False*. The negation of *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 2.1b for an example. On Lifeline *L3*, the sending OS of Message *m6* takes the negation of *isCritical* for the EU on Lifeline *L3* as an enabling condition.

Figure 2.1b shows an example Sequence Diagram with nested CFs, *i.e.*, a Parallel containing a Critical Region. The implementation of its main module and the modules of Lifeline *L2* and its CEUs and EUs are shown in figure 6.3. In the module of Lifeline *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, *op1*, is located on *L2*. Thus, *cond1* for *op1* is initialized and declared in the Parallel's CEU module on *L2*. It is set to the value of the evaluation step and remains that value in the following steps. Variable *op1_eva* is initialized to *-1*, and then is set depending on the value of *cond1* when entering the CEU, *i.e.*, it is set to *1* if *cond1* evaluates to *True* or *0* otherwise. In each EU module of the Parallel, a variable *chosen* is used to denoted whether the EU is chosen to execute OSs.

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

```
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_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.enabled & !l_L2.enabled & !l_L3.enabled))
    & (l_L1.chosen -> l_L1.enabled)
    & (l_L2.chosen -> l_L2.enabled)
    & (l_L3.chosen -> l_L3.enabled))
INVAR
    ((( l_L1.CF1.op1.chosen & !l_L1.CF1.op2.chosen)
    | (!l_L1.CF1.op1.chosen &  l_L1.CF1.op2.chosen)
    | (!l_L1.CF1.op1.enabled & !l_L1.CF1.op2.enabled))
    & (l_L1.CF1.op1.chosen -> l_L1.CF1.op1.enabled)
    & (l_L1.CF1.op2.chosen -> l_L1.CF1.op2.enabled))
INVAR
    ((( l_L2.CF1.op1.chosen & !l_L2.CF1.op2.chosen)
    | (!l_L2.CF1.op1.chosen &  l_L2.CF1.op2.chosen)
    | (!l_L2.CF1.op1.enabled & !l_L2.CF1.op2.enabled))
    & (l_L2.CF1.op1.chosen -> l_L2.CF1.op1.enabled)
    & (l_L2.CF1.op2.chosen -> l_L2.CF1.op2.enabled))
INVAR
    ((( l_L3.CF1.op1.chosen & !l_L3.CF1.op2.chosen)
    | (!l_L3.CF1.op1.chosen &  l_L3.CF1.op2.chosen)
    | (!l_L3.CF1.op1.enabled & !l_L3.CF1.op2.enabled))
    & (l_L3.CF1.op1.chosen -> l_L3.CF1.op1.enabled)
    & (l_L3.CF1.op2.chosen -> l_L3.CF1.op2.enabled))

MODULE L2(L1, L3)
 VAR
  OS_r1 : boolean;
  OS_r7 : boolean;
  state : {sinit, r1, r7};
  CF1 : par_L2(state, chosen, L1.CF1, L3.CF1);
  chosen : boolean;
 DEFINE
  r1_enabled := state = sinit & (L1.state = s1
               | L1.state = s7);
  r7_enabled := state = r1 & CF1.flag_final
              & L1.state = s7;
  enabled  := r1_enabled | r7_enabled | CF1.enabled;
  flag_final := state = r7;
```

```
 ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_r1)  :r1;
    state = r1 & next(OS_r7)     :r7;
    1                            :state;
  esac;
  init(OS_r1) := FALSE;
  next(OS_r1) := case
    chosen & r1_enabled     :TRUE;
    OS_r1                   :FALSE;
    1                       :OS_r1;
  esac;
  init(OS_r7) := FALSE;
  next(OS_r7) := case
    chosen & r7_enabled     :TRUE;
    OS_r7                   :FALSE;
    1                       :OS_r7;
  esac;
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;
```

Figure 6.3: NuSMV module for Parallel

```
MODULE par_op1_L2(L2_chosen, par_L1_op1, par_L3_op1,
                                             op1_eva)
 VAR
  OS_s2 : boolean;
  state : {sinit, s2};
  CF2 : critical_L2(state, chosen, L2_chosen,
                           par_L3_op1.CF2);
  chosen : boolean;
 DEFINE
  s2_enabled := state=sinit & op1_eva=1;
  enabled := s2_enabled | CF2.enabled;
  flag_final := (state=s2 & CF2.flag_final & op1_eva=1)
                                            | op1_eva=0;
 ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_s2)     :s2;
    1                               :state;
  esac;
  init(OS_s2) := FALSE;
  next(OS_s2) := case
    chosen & L2_chosen & s2_enabled :TRUE;
    OS_s2                           :FALSE;
    1                               :OS_s2;
  esac;
MODULE critical_L2(state, chosen, L2_chosen, critical_L3)
 VAR
  op1 : critical_op1_L2(chosen, L2_chosen,
        critical_L3.op1, critical_L3.op3_eva, state);
 DEFINE
  enabled := op1.enabled;
  flag_final := op1.flag_final;
MODULE critical_op1_L2(chosen, L2_chosen,
       critical_L3_op1, op3_eva, pre_state)
 VAR
  OS_r3 : boolean;
  OS_s4 : boolean;
  state : {sinit, r3, s4};
  isCritical : boolean;
 DEFINE
  r3_enabled  := state=sinit & op3_eva=1 & pre_state=s2 &
   (critical_L3_op1.state=s3 | critical_L3_op1.state=r4);
  s4_enabled  := state = r3;
  enabled := r3_enabled | s4_enabled;
  flag_final := (state = s4 & op3_eva=1) | op3_eva=0;
```

```
ASSIGN
 init(state) := sinit;
 next(state) := case
   state = sinit & next(OS_r3)      :r3;
   state = r3 & next(OS_s4)         :s4;
   1                                :state;
 esac;
 init(OS_r3) := FALSE;
 next(OS_r3) := case
   chosen & L2_chosen & r3_enabled :TRUE;
   OS_r3                            :FALSE;
   1                                :OS_r3;
 esac;
 init(OS_s4) := FALSE;
 next(OS_s4) := case
   chosen & L2_chosen & s4_enabled :TRUE;
   OS_s4                            :FALSE;
   1                                :OS_s4;
 esac;
 init(isCritical) := FALSE;
 next(isCritical) := case
   next(state) = r3                 :TRUE;
   next(state) = s4                 :FALSE;
   1                                :isCritical;
 esac;
```

Figure 6.4: NuSMV module for Critical Region

6.4.3 Branching

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 *exe* indicates the execution status of the applicable Operand, *i.e.*, *exe* is set to *True* if the Operand is chosen to execute. The variable *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 *exe* can be set to *True* only if the Operand's *cond* evaluates to *True*. It also restricts that at most one Operand can be chosen to execute, *i.e.*, at most one *exe* can be set to *True* at a time, or all Operand Constraints evaluate to *False*. The use of *exe* guarantees that all the enclosed Lifelines choose the same Operand's EU module to execute to avoid inconsistent choices (*e.g.*, Lifeline *L1* chooses Operand *1*'s EU whereas Lifeline *L2* chooses Operand *2*'s EU). The *cond* of the chosen Operand stays *True* and those of the unchosen Operands are set to *False* and stay *False*.

Figure 4.10 is an example of an Alternatives with three Operands enclosing three Lifelines. Figure 6.5 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.

```
MODULE main
...
 INVAR
   ((l_L2.CF1.exe1->l_L2.CF1.cond1)
   &(l_L2.CF1.exe2->l_L2.CF1.cond2)
   &(l_L2.CF1.exe3->l_L2.CF1.cond3))
 & ((l_L2.CF1.exe1 & !l_L2.CF1.exe2 &
    !l_L2.CF1.exe3)
   |(!l_L2.CF1.exe1 & l_L2.CF1.exe2 &
     !l_L2.CF1.exe3)
   |(!l_L2.CF1.exe1 & !l_L2.CF1.exe2 &
     l_L2.CF1.exe3)
   |(!l_L2.CF1.cond1 & !l_L2.CF1.cond2 &
     !l_L2.CF1.cond3))
   ...
MODULE alt_L2(state, chosen, alt_L3)
 VAR
  op1 : alt_op1_L2(op1_eva, chosen, alt_L3.op1);
  op2 : alt_op2_L2(op2_eva, chosen, alt_L3.op2);
  op3 : alt_op3_L2(op3_eva, chosen, alt_L3.op3);
  op1_eva : -1..1;
  op2_eva : -1..1;
  op3_eva : -1..1;
  cond1 : boolean;
  cond2 : boolean;
  cond3 : boolean;
  exe1  : boolean;
  exe2  : boolean;
  exe3  : boolean;
 DEFINE
  enabled := op1.enabled | op2.enabled | op3.enabled;
  flag_final := op1.flag_final & op2.flag_final
                              & op3.flag_final;
 ASSIGN
  init(op1_eva)  := -1;
  next(op1_eva)  := case
    op1_eva = -1 & next(state) = r1 & !next(exe1)  :0;
    op1_eva = -1 & next(state) = r1 &  next(exe1)  :1;
    1                                             :op1_eva;
  esac;
  ...
  init(cond1)   := {TRUE, FALSE};
  next(cond1)   := case
    op1_eva = -1           : {TRUE, FALSE};
    op1_eva = 0            : FALSE;
    op1_eva = 1            : TRUE;
    1                      : cond1;
  esac;
  ...
```

```
init(exe1)    := {TRUE, FALSE};
next(exe1)    := case
  op1_eva = -1           : {TRUE, FALSE} ;
  op1_eva != -1          : exe1;
  1                      : exe1;
esac;
...
```

Figure 6.5: NuSMV module for Alternatives

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 6.4. We demonstrate an example of an Option in figure 4.8. Figure 6.6 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 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.

Representing Break

The Break has been rewritten to an Alternatives with two Operands as we describe in section 4.2.2. Therefore, the Break can be mapped to NuSMV modules as an Alternatives.

```
MODULE opt_L2(state, chosen, opt_L1, opt_L3)
 VAR
  flag_opt : -1..1;
  op1 : opt_op1_L2(chosen, opt_L1.op1, opt_L3.op1,
                                            op1_eva);
  cond1 : boolean;
  op1_eva : -1..1;
 DEFINE
  flag_final := op1.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;

MODULE opt_op1_L2(chosen, opt_L1_op1, opt_L3_op1,
                                            op1_eva)
 VAR
  OS_s2 : boolean;
  state : {sinit, s2};
 DEFINE
  s2_enabled  := state = sinit & op1_eva = 1;
  enabled := s2_enabled;
  flag_final := (state=s2 & op1_eva=1) | op1_eva=0;
 ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_s2)       :s2;
    1                                 :state;
   esac;
  init(OS_s2) := FALSE;
  next(OS_s2) := case
    chosen & s2_enabled               :TRUE;
    OS_s2                             :FALSE;
    1                                 :OS_s2;
   esac;
   ...
```

Figure 6.6: NuSMV module for Option

6.4.4 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, *i.e.*, OSs in iteration *n* take place before OSs in iteration *n+1*.

For example, the NuSMV module in figure 6.7 implements the EU of the Loop's Operand on Lifeline *L1* in figure 4.15, with three iterations. OSs *s1* and *r3* are mapped to two arrays of three boolean variables, *i.e.*, the unfolded EU contains six OSs. The variable *state* has a value for each OS to record its execution, *e.g.*, the value of *state* is set to *s1_2*, representing OS *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, *e.g.*, *OS_s1[3]* representing *s1* in the third iteration, which is enabled only if *r3* in the second iteration (*OS_r3[2]*) has executed.

We also translate the bounded Loop, whose *maxint* is given, to NuSMV model. To keep each OS and Constraint within different iterations of a Loop unique, one way to implement an OS or a Constraint is defining an array to rename the OS or the Constraint of each iteration. For each Lifeline, We use n to represent the current iteration number. In this way, an OS within the Loop's Operand, OS_r1, and Constraint *cond* in iteration n can be represented as $OS\_r1[n]$ and $cond[n]$ respectively. For example, if a Loop iterates at most three iterations, OS_r1 in different iterations are defined as $OS\_r1[1]$, $OS\_r1[2]$ and $OS\_r1[3]$.

We need to evaluate the Interaction Constraint of its sole Operand after minimum number of iterations. If $n \leq minint$, the Loop executes. If $minint < n \leq maxint$, the Loop executes only if $cond[n]$ evaluates to $True$. Otherwise, the Loop terminates and the values of the Constraint of remaining iterations (*i.e.*, from $cond[n+1]$ to $cond[maxint]$) set to *False*. The Loop no longer executes when its iteration reaches $maxint$.

```
MODULE loop_op1_L1(chosen, op1_L2, op1_L3, op1_eva)
 VAR
  OS_s1:array 1..3 of boolean;
  OS_r3:array 1..3 of boolean;
  state:{sinit, s1_1, r3_1, s1_2, r3_2, s1_3, r3_3};
 DEFINE
  s1_1_enabled:= state = sinit & op1_eva=1;
  r3_1_enabled:= state = s1_1&(op1_L3.state = s3_1
         |op1_L3.state = r2_2|op1_L3.state = s3_2
         |op1_L3.state = r2_3|op1_L3.state = s3_3);
  s1_2_enabled:= state = r3_1;
  r3_2_enabled:= state = s1_2&(op1_L3.state = s3_2
         |op1_L3.state = r2_3|op1_L3.state = s3_3);
  s1_3_enabled:= state = r3_2;
  r3_3_enabled:= state = s1_3&(op1_L3.state = s3_3);
  enabled:=s1_1_enabled|r3_1_enabled|s1_2_enabled
          |r3_2_enabled|s1_3_enabled|r3_3_enabled;
  flag_final:= (state = r3_3 & op1_eva=1) | op1_eva=0;
 ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_s1[1])    :s1_1;
    state = s1_1 & next(OS_r3[1])     :r3_1;
    state = r3_1 & next(OS_s1[2])     :s1_2;
    state = s1_2 & next(OS_r3[2])     :r3_2;
    state = r3_2 & next(OS_s1[3])     :s1_3;
    state = s1_3 & next(OS_r3[3])     :r3_3;
    1                                  :state;
   esac;
  init(OS_s1[1]) := FALSE;
  next(OS_s1[1]) := case
    chosen & s1_1_enabled       :TRUE;
    OS_s1[1]                    :FALSE;
    1                           :OS_s1[1];
   esac;
  init(OS_r3[1]) := FALSE;
  next(OS_r3[1]) := case
    chosen & r3_1_enabled       :TRUE;
    OS_r3[1]                    :FALSE;
    1                           :OS_r3[1];
   esac;
  init(OS_s1[2]) := FALSE;
  next(OS_s1[2]) := case
    chosen & s1_2_enabled       :TRUE;
    OS_s1[2]                    :FALSE;
    1                           :OS_s1[2];
   esac;
```

```
  init(OS_r3[2]) := FALSE;
  next(OS_r3[2]) := case
    chosen & r3_2_enabled       :TRUE;
     OS_r3[2]                   :FALSE;
     1                          :OS_r3[2];
    esac;
  init(OS_s1[3]) := FALSE;
  next(OS_s1[3]) := case
    chosen & s1_3_enabled       :TRUE;
    OS_s1[3]                    :FALSE;
    1                           :OS_s1[3];
   esac;
  init(OS_r3[3]) := FALSE;
  next(OS_r3[3]) := case
    chosen & r3_3_enabled       :TRUE;
    OS_r3[3]                    :FALSE;
    1                           :OS_r3[3];
   esac;
```

Figure 6.7: NuSMV module for Loop

6.4.5 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 *flag_final* of the preceding EU on the same Lifeline as an enabling condition, *i.e.*, the EU can not execute before the preceding one completes.

Figure 4.20 is an example of a Weak Sequencing, whose EU of the second Operand on Lifeline *L2* is mapped into a NuSMV module (see figure 6.8). In the EU module, the first OS, *r4*, has an enabling condition , which is the variable *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 *L2* can be enforced.

The semantics of the Strict Sequencing enforces the total order between adjacent Operands. An EU module of an Operand (other than the first one) within a Strict Sequencing takes the

```
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;
  ...
```

Figure 6.8: NuSMV module for Weak Sequencing

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 4.22 is an example of a Strict Sequencing and figure 6.9 shows the EU module of the second Operand on Lifeline $L2$. The OS *r4* takes the variables *flag_final*, one for each EU of the first Operand as enabling conditions to enforce the total order among Operands.

```
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_L1_op1.flag_final
               & 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;
  ...
```

Figure 6.9: NuSMV module for Strict Sequencing

6.4.6 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 types of Messages 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 list of types of Messages which should be considered within the Consider. 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, then the Messages of the type can not occur within the Consider. For example, the Consider in figure 4.27 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 ICF, 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 and stays *1* in the following steps. In each EU module of the ICF, 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 4.26 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 6.10). 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 CCF, each OS of the considered type Messages can be defined as a boolean variable in the EU module of the Consider on the Lifeline where it is located. If the

```
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_L2.s3_chosen = 1
                                         & op1_eva=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
    r3_chosen=0 & L3_chosen & r3_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;
  ...
```

Figure 6.10: NuSMV module for Ignore

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 in an Ignore. For example, figure 6.11 shows an EU module on Lifeline *L2* for the Consider in figure 4.27. Message *m5* is considered but does not appear in the Consider, 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 CCF, each OS of the Messages not considered by Consider and other OSs are interleaved, which is represented by INVAR statements.

6.4.7 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*.

6.4.8 General Ordering

A General Ordering enforces the order between two unordered OSs, which describes that one 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.

6.5 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

```
MODULE consider_op1_L2(L2_chosen, op1_L3, op1_eva)
 VAR
  OS_s2 : boolean;
  OS_s3 : boolean;
  state : {sinit, s2, s3};
  chosen : boolean;
  OS_s6 : boolean;
  s6_chosen : {-1, 0, 1};
 DEFINE
  s2_enabled  := state = sinit & op1_eva = 1;
  s3_enabled  := state = s2;
  enabled := s2_enabled | s3_enabled;
  flag_final := (state = s3 & op1_eva=1) | op1_eva = 0;
  s6_enabled  := s6_chosen != 1 & ((op1_eva != 1) |
       (op1_eva = 1 & flag_final & op1_L3.flag_final));
  OS_s5 := FALSE;
 ASSIGN
  init(state) := sinit;
  next(state) := case
    state = sinit & next(OS_s2)      :s2;
    state = s2 & next(OS_s3)         :s3;
    1                                :state;
  esac;
  init(OS_s2) := FALSE;
  next(OS_s2) := case
    chosen & L2_chosen & s2_enabled      :TRUE;
    OS_s2                                :FALSE;
    1                                    :OS_s2;
  esac;
  init(OS_s3) := FALSE;
  next(OS_s3) := case
    chosen & L2_chosen & s3_enabled      :TRUE;
    OS_s3                                :FALSE;
    1                                    :OS_s3;
  esac;
  init(OS_s6) := FALSE;
  next(OS_s6) := case
    s6_chosen = 0 & L2_chosen & s6_enabled  :TRUE;
    OS_s6                                   :FALSE;
    1                                       :OS_s6;
  esac;
  init(s6_chosen) := {-1, 0};
  next(s6_chosen) := case
    s6_chosen = -1              :{-1, 0};
    next(OS_s6)                 :1;
    1                           :s6_chosen;
  esac;
  ...
```


Figure 6.11: NuSMV module for Consider

Interaction. If the Interaction contains CFs, each of its CEUs is mapped to a CEU module and the EUs 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.

6.6 Proof for NuSMV Model of Sequence Diagram with Combined Fragments

We wish to prove that the NuSMV model for a Sequence Diagram with CFs capture the semantics of the Sequence Diagram. Recall that the semantic rules general to all CFs are shown in section 2.2. The semantics of each CF Operator is shown in section 2.3. Section 6.4 describes the mapping strategy for Sequence Diagram with CFs, where an example of NuSMV model for a Sequence Diagram with CFs is shown in figure 6.3.

Lemma 6.15. *A given Sequence Diagram with CFs, seq, directly contains h Message. In the CFs, p Messages are enclosed in Operands whose Interaction Constraints evaluate to True,* i.e.*, if a Message is enclosed in multiple nested Operands, all the Interaction Constraints of the Operands evaluate to True. For other q Messages within the CFs, each Message is enclosed in one Operand or multiple nested Operands, where at least one Operand's Interaction Constraint evaluate to False. If $\sigma \in (\Sigma^{seq}_{NuSMV})^{\omega}$, then σ must have the form, $\sigma = \sigma_{[1..2h+2p]} \cdot \tau^{\omega}$, where $\sigma_{[1..2h+2p]}$ contains no τ.*

Proof. In the NuSMV model for seq, M_{seq}, the INVAR statement asserts that one enabled OS on one Lifeline can take place in each step until no more OSs are enabled. Therefore, no τ occurs between two OSs in σ. In the Lifeline modules, the variables of OSs define that each OS directly enclosed in seq can execute once and only once. In the CEU modules, variables op_eva of the EUs whose Constraints evaluate to *True* set to 1, indicating that the OSs within the EUs can be enabled to execute. Otherwise, variable op_eva sets to 0, indicating that the OSs within the EUs whose Constraints evaluate to *False* cannot be enabled to execute. The variables of OSs in the EU modules define that each enabled OS can execute once and only once. Therefore, we can infer that σ have the form, $\sigma = \sigma_{[1..2h+2p]} \cdot \tau^{\omega}$, where $\sigma_{[1..2h+2p]}$ does not contain τ. $\square$

A given Sequence Diagram, seq_r, directly contains k Lifelines, h Messages and r CFs, which contain $p + q$ Messages. Each CF does not contain other CFs. For the Messages within the CFs, p Messages are enclosed in Operands whose Interaction Constraints evaluate to *True*, while q Message are enclosed in Operands whose Interaction Constraints evaluate to *False*.

We wish to prove that, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_r})^*, \upsilon \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq_r})^{\omega}$. The semantic rules of seq_r define that each OS which is directly enclosed in seq_r or an Operand whose Constraint evaluates to *True*, occurs once and only once. Thus, $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_r})^*, |\upsilon| = 2h + 2p$. From lemma 6.15, we learn that $\forall \sigma.\sigma \in (\Sigma_{NuSMV}^{seq_r})^{\omega}, \sigma = \sigma_{[1..2h+2p]} \cdot \tau^{\omega}$, where $\sigma_{[1..2h+2p]}$ contains no τ. $\sigma_{[1..2h+2p]} \in PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^{\omega})$. If $\forall \upsilon.\upsilon \in (\Sigma_{sem}^{seq_r})^*, \upsilon \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq_r})^{\omega}$, we can infer that, $\upsilon \in PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^{\omega})$, *i.e.*, $(\Sigma_{sem}^{seq_r})^* \subseteq PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^{\omega})$.

We also wish to prove that $\forall \sigma.\sigma \in (\Sigma_{NuSMV}^{seq_r})^{\omega}, \sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_r})^*$, *i.e.*, $PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^{\omega}) \subseteq (\Sigma_{sem}^{seq_r})^*$.

Theorem 6.16. *$(\Sigma_{sem}^{seq_r})^*$ and $PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^{\omega})$ are equal.*

We provide the proof of theorem 6.16 in appendix B.4.

Chapter 7: TOOL SUITE IMPLEMENTATION

As a proof-of-concept, we have developed a tool suite, implementing the techniques described in this dissertation. Recall figure 7.1 illustrates the architecture of the software framework.

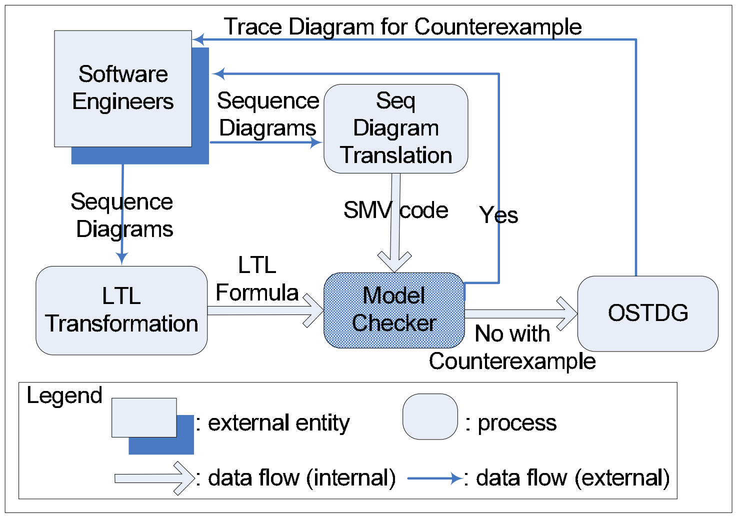


Figure 7.1: Architecture of tool suite

The software engineer uses MagicDraw to create a Sequence Diagram, which can be converted to a textual representation in terms of XML using our MagicDraw plugin. The Sequence Diagram Translation tool and the LTL Transformation Tool take the XML representation as input, parses it into a syntax tree, and transforms it into a NuSMV model and a LTL formula respectively. NuSMV model checker takes as input the generated NuSMV model and a temporal logic formula that is translated automatically from a Sequence Diagram or specified by the software engineer. 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 a Sequence Diagrams using NuSMV, staying solely within the notation realm of Sequence Diagrams.

Our tool suite consists of four components, which include an automated tool translating Sequence Diagrams with CFs into LTL formulas, an automated tool translating Sequence Diagrams with CFs into NuSMV modules, an OSTDG tool, and a user interface. Our tool suite are implemented using Java as described in the following sections.

7.1 Translating Sequence Diagram into LTL Formulas

The LTL Transformation Tool is a tool for translation Sequence Diagrams with CFs into LTL formulas. It is typically used to generate the LTL properties which are described using Sequence Diagrams by user. It is designed for users who do not have strong background of temporal logic to ease their efforts for analyzing their requirements. The tool supports all 12 CFs, nested CFs, and Interaction Use. The scope of the tool is defined in chapter 4.5.3, *e.g.*, the tool only supports complete Messages.

This tool accepts the XML representation of a Sequence Diagram, which is converted by our MagicDraw plugin. We write a parser (*i.e.*, class parseXMLFile) to parse the XML representation and create the syntax tree of the Sequence Diagram. We define multiple classes to indicate the structure of a Sequence Diagram, which consists of:

- Class SD: defines the structure of a Sequence Diagram, which contains a set of Lifelines and a set of CFs.
- Class Lifeline: defines the structure of a Lifeline, which contains a set of OSs not enclosed in any CEUs and a set of CEUs not enclosed in other CEUs.
- Class CF: defines the structure of a Combined Fragment, which contains its Interaction Operator, a set of Lifelines enclosed in the CF, and a set of Interaction Operands.
- Class Operand: defines the structure of an Interaction Operand.
- Class OS: defines the structure of an Occurrence Specification.
- Class CEU: defines the structure of a CEU.

- Class EU: defines the structure of an EU.

- Class Cond: defines the structure of a condition.

- Class Elements: the OSs and CEUs on a Lifeline are called elements of the Lifeline. This class helps to create the list of OSs and CEUs on the same Lifeline.

We implement class Translate2LTL to translate the syntax tree of the Sequence Diagrams into LTL formulas. The translation strategy is based on our LTL templates in chapter 4. In our LTL templates, we evaluate the Interaction Constraints of each Interaction Operand using auxiliary functions. To implement these auxiliary functions, we need to determine when to evaluate the Interaction Constraints. However, for a Sequence Diagram, the value of the Interaction Constraints in it can be modified by other Sequence Diagrams of the same system. Thus, we need to evaluate the Interaction Constraint of an Interaction Operand right before entering the Combined Fragment which contains the Interaction Operand, *i.e.*, the Interaction Operand is ready to execute. The value of the Interaction Constraint is maintained after evaluation to keep the execution of the Operand is consistent. We have implemented the auxiliary functions which are related to the Interaction Constraints, *e.g.*, $TOP(u)$, $TBEU(u)$, *etc.*. We demonstrate and explain the modified LTL templates in appendix C.

We start with relating each OS with Interaction Constraints. For an Operand, its Interaction Constraint is associated with all the OSs within it. If an OS is enclosed in multiple nested CFs, the Interaction Constraints of all the Operands enclosing the OS are associated with the OS. The Interaction Constraint of the OS which are directly enclosed in the Sequence Diagram is considered as evaluating to $True$.

We translate the structure of the Sequence Diagram into the sub-formulas of our modified LTL templates. First, we consider the sub-formulas for BEUs which are directly enclosed in a Sequence Diagram with CFs. These sub-formulas can also be applied to translate basic Sequence Diagrams. To obtain sub-formula α_g for BEU g, we need to represent (1) the order of each pair of adjacent OSs in g; (2) each OS in g must happen once and only once. To obtain sub-formula β_j for Message

j, we represent the order between its sending and receiving OSs. For the Sequence Diagram seq, we need to obtain all the OSs from its enclosing Lifelines and translate the interleaving semantics as the definition of sub-formula $\bar{\varepsilon}_{seq}$. $\bar{\varepsilon}_{seq}$ also enforces that each OS must happen if the Interaction Constraints associated with it evaluate to *True*.

Then, we consider the sub-formulas which are general to all CFs. In the modified sub-formula $\bar{\Phi}^{CF}$, each Operand is translated into a conjunct, which represents the evaluation of the Operand's Interaction Constraint, and the order of OSs within the Operand. We evaluate the Interaction Constraint of an Operand when the OSs locate before the CF have executed, where these OS should locate on the Lifeline where the Interaction Constraint locates on. If the Constraint evaluates to *False*, the Constraint keeps *False* and the Constraints of the nested set to *False*. Otherwise, the Constraint keeps *True* and the order of OSs directly enclosed in the Operand are enforced using sub-formula $\bar{\theta}^{m}$. The nested CFs are translated into $\bar{\Phi}^{CF}$ recursively. $\bar{\theta}^{m}$ defines the order of OSs within m using modified $\bar{\alpha}_g$ and $\bar{\beta}_j$, which specify the OSs order for every state. The order of OSs in BEU g is translated into $\bar{\alpha}_g$, and the order of OSs of Message j is translated into $\bar{\beta}_j$. For each CF, the order between the preceding/succeeding set of OSs and the OSs within the CF is translated into $\bar{\gamma}^{CF}$. We implement the algorithm (see appendix A) to calculate the preceding/succeeding set OSs of the CF. The order between the preceding set of OS and the succeeding set of OSs is translated into $\bar{\eta}^{CF}$. We have discussed the first OS occurring with an Operand in chapter 4.5.3. For each Operand of a CF, the order between the first OS occurring the Operand and other OSs is translated into $\bar{\mu}^{CF}$.

Finally, we consider the sub-formulas which are specific for each CF. We rewrite the sub-formula $\bar{\Phi}^{CF}$ using $\bar{\Phi}^{CF}_{alt}$ and $\bar{\Phi}^{CF}_{loop}$ for representing the semantics of Alternatives and Loop respectively. An Alternatives is translated into $\bar{\Phi}^{CF}_{alt}$ as a general CF with an additional sub-formula ϑ^{CF}. For each Operand, ϑ^{CF} relates its Constraint and its *exe*, which indicates that if the Operand is chosen to execute. Each OS within Alternatives is also associated with the *exe* to indicate its execution. We translate all iterations of Loop into $\bar{\Phi}^{CF}_{loop}$, where these iterations are connected using Weak Sequencing. Operand m of a Critical Region is translated into $\bar{\theta}^{m}_{critical}$, which conjuncts sub-formula $\bar{\delta}_{M1,M2}$ with $\bar{\theta}^m$. $\bar{\delta}_{M1,M2}$ enforces that the execution of the OSs within the Critical Region's EU on each Lifeline cannot be interrupted by other OSs on the same Lifeline. Similarly, the Operand of Assertion is translated into $\bar{\theta}^m_{assert}$, whose sub-formula $\bar{\lambda}^{i,seq}_{M1,M2}$ asserts that, for each Lifeline, the execution of the preceding set of OSs and the OSs within EU of the Assertion cannot interleaved by other OSs. We translate other CFs into LTL sub-formulas as our LTL templates. The LTL formulas are printed into a file using Class printOut.

7.2 Translating Sequence Diagram into NuSMV Model

The aim of Sequence Diagram Translation tool is translating UML Sequence Diagrams into the input language of NuSMV. This tool is used to transform a Sequence Diagram which needs to be analyzed into a NuSMV model. It helps software engineers to analyze their requirements automatically. The scope of this tool is the same as the LTL Transformation Tool. The tool accepts the XML representation of a Sequence Diagram too. To preprocessing the Sequence Diagram, we can use the parser we developed before to create the syntax tree, which contains OSs, CFs and other components as we defined.

The translation from the syntax tree of a Sequence Diagram into NuSMV models is implemented using class Translate2SMV. We have presented the translation strategy in chapter 6. We preserve the structure of a Sequence Diagram when we map it into NuSMV models. First, we describe the Sequence Diagram using a main module, where the Lifelines within the Sequence Diagram are instantiated and declared as module instances. Each Lifeline module instance takes other Lifeline module instances as parameters. We represent the interleaving semantics among Lifeline using INVAR statements as defined in our mapping strategy. It takes all Lifeline modules instances of the Sequence Diagram and describes that only one Lifeline is chosen to execute until the Sequence Diagram finishes execution. In the main module, we also express the interleaving semantics among EUs in Parallel or OSs in Consider/Ignore using INVAR statements, which will be discussed when we translate the CFs into NuSMV modules. We map the OSs as boolean variables in the Lifeline/EU module which directly encloses them. To access the OS variable, we need to append the name of every Lifeline/EU which encloses the OS with **.**(DOT) before the variable name. In the main module, we define each OS variable using a derived variable to indicate the access of the variable. For example, we define s_1 of Lifeline module $L1$ as $s1 := l_L1.s1$; in the main module. This procedure makes the OS variable names in the NuSMV model and the ones in the LTL formulas are consistent.

Then, we translate each Lifeline into a NuSMV module. In VAR section, the NuSMV module takes the directly enclosed OSs, the directly enclosed CEUs, *state*, and *chosen* as variables. Each directly enclosed OS is mapped to a boolean variable. Each directly enclosed CEU is instantiated and declared as a module instance, which carries related CEUs, *state* and *chosen* as parameters. *state* is an enumerated type variable, which has an initial value an a value for each OS to record the execution status. *chosen* is a boolean variable, which is used to specify the interleaving semantics in main module. If the Interaction Operator of any directly enclose CEU is *critical* or *assert*, the Lifeline module contains variable *isCritical* or *isAssertion* for the CEU. We will discuss it when translating the CFs into NuSMV modules. In DEFINE section, a derived variable, *enabled*, is defined for each OS to indicate the OS is ready to execute. Another *enabled* is defined for the Lifeline to indicate the Lifeline module is ready to execute. A derived variable, *final*, is defined for the Lifeline to indicate the Lifeline module reaches final state. In ASSIGN section, the execution of the Lifeline module is mapped to the transition relation of variable *state*. The execution of each OS is mapped to the transition relation of its variable.

Next, we map each CEU directly enclosed in the Lifelines into a NuSMV module. A CEU consists of one or more EUs and an Interaction Operator. In VAR section, each EU is instantiated and declared as a module instance, which carries the related EU instances, *chosen*, and *op_eva* as parameters. For the EU's Interaction Constraints which locate on the Lifeline, each Constraint is mapped to boolean variables *cond* and *op_eva*. *cond* records the value of the Constraint, and *op_eva* records the execution status of the Operand. If the CEU's Operator is *alt*, each Constraint is also mapped to boolean variable *exe* to indicate the Operand is chosen to execute. In DEFINE section, a derived variable, *enabled*, is defined to indicate the CEU is ready to execute. Variable *final* is defined to indicate the CEU reaches its final state. In ASSIGN section, *cond* shows the evaluation of the Constraint. The transition relation of *op_eva* captures the status that the EU is ready to execute, the EU will not execute, or the EU states to execute.

Finally, we map each EU of the CEUs into a NuSMV module. The EU module is quite similar to the Lifeline module. We can reuse some methods for Lifeline modules to translates each EU into an EU module. Additional variables may be introduced for EU modules with different Interaction Operators. For EU whose Operator is *par*, a variable, *chosen* is defined for the EU module to indicate that the EU can be chosen to execute. We have mapped the interleaving semantics of Parallel to INVAR statements in the main module. For EU whose Operator is *critical*, the transition relation of variable *isCritical* is defined in ASSIGN section. *isCritical* constraints the CEU of Critical Region cannot be interleaved by other OSs on the same Lifeline. For EU whose Operand is *loop*, each OS or Constraint is defined as an array. The *state* and derived variables are changed correspondingly. For EU whose Operand is *assert*, the transition relation of variable *isAssertion* is defined in ASSIGN section. For EU whose Operand is *weak* or *strict*, variable *enabled* for the first OS takes extra enabling condition to represent the order between EUs. For EU whose Operand is *consider* or *ignore*, each ignored OS is mapped to a boolean variable. We also introduce an enumerated type variable, *chosen*, for each ignored OS. The interleaving semantics between the ignored OSs and other OSs is represented by INVAR statement in main module. The nested CEUs and EUs are mapped into NuSMV modules recursively by following the same procedure. The generated NuSMV modules are printed to file using Class printOut.

7.3 Occurrence Specification Trace Diagram Generator (OSTDG) Tool

We develop the OSTDG to generate trace diagram visualizations of counterexamples produced by the NuSMV model checker. Trace diagram is similar to basic Sequence Diagram with asynchronous Message. It contains one or more Lifelines, which are communicated using Message.

For the counterexample, we can get the information of Lifelines from the first state, and generate the Lifeline in the trace diagram. Every state change is triggered by the execution of an OS until all OSs have executed. We can find the executing OS and the chosen Lifeline for each state, and generate the OS on the corresponding Lifeline in the trace diagram. Each OS should locate below its preceding OS. Later, we connect the OSs of the same Message using arrow from the sending OS to the receiving OS, and add the Message name near the arrow. An example of trace diagram is shown in figure 7.2.

7.4 User Interface

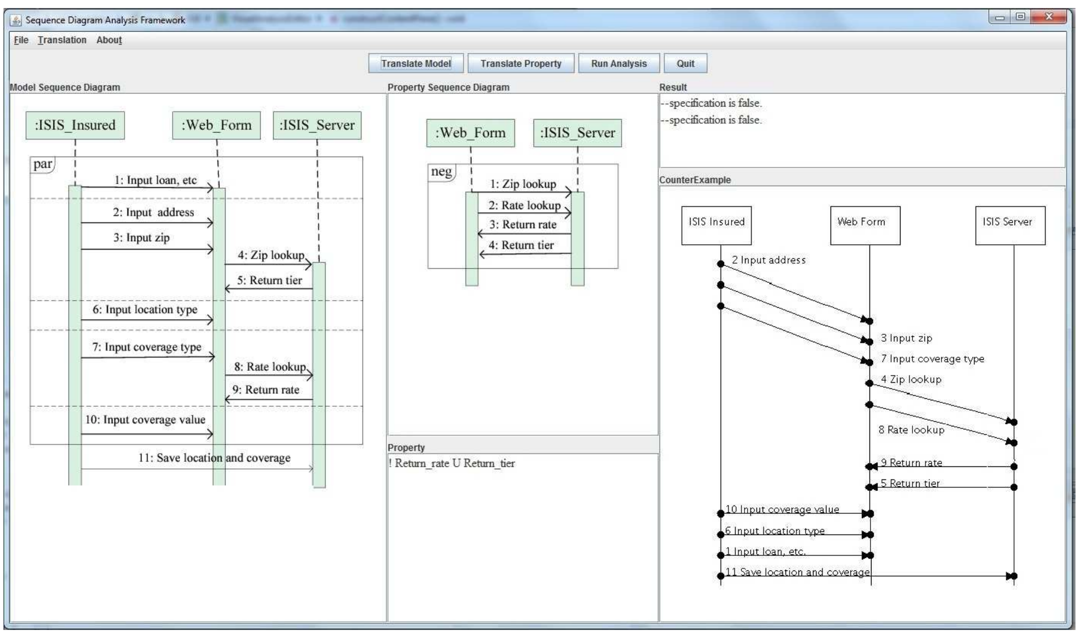


Figure 7.2: Screenshot of the tool suite (Case Study 1).

In previous sections, we have introduced the components of our tool suite. We build a user interface to integrate these components and ease users' effort to utilize our tool suite. Figure 7.2 shows a screenshot of the user interface, which contains of three columns. The left column shows the Sequence Diagram to be checked, which is translated to a NuSMV model. The user can load a Sequence Diagram by choosing the item $LoadModelSequenceDiagram$ in the menu of $File$, and translate it into a NuSMV model by clicking the button $TranslateModel$ or choosing the item $TranslateModel$ in the menu of $Translation$. The middle column shows the properties, which can be derived from a Sequence Diagram (the upper box), or specified by the software engineer (the lower box). The user can load a Sequence Diagram by choosing the item $LoadPropertySequenceDiagram$ in the menu of $File$, and translate it into LTL formulas by clicking the button $TranslateProperty$ or choosing the item $TranslateProperty$ in the menu of $Translation$. If the user would like to input some properties as LTL formulas, he can choose the item $AddProperty$ in the menu of $File$ and type the formulas. With the NuSMV model and LTL properties, the user can click the button $RunAnalysis$ to verify the model against the properties. The result of verification is shown in the right column. If the model does not satisfy a property, a counterexample is generated by the NuSMV model checker. To read the counterexample, the user can click the item $GenerateOSTD$ in the menu of $Translation$. The trace diagram of the counterexample will be shown in the right column. Our user interface demonstrates the trace diagram for one counterexample at a time. If multiple properties are checked together, the user needs to click $GenerateOSTD$ again to see the trace diagram for the counterexample of the next property. Figure 7.2 is a screenshot of the user interface when we run the case study example 1, while figure 7.3 is a screenshot of the user interface when we run the case study example 2.

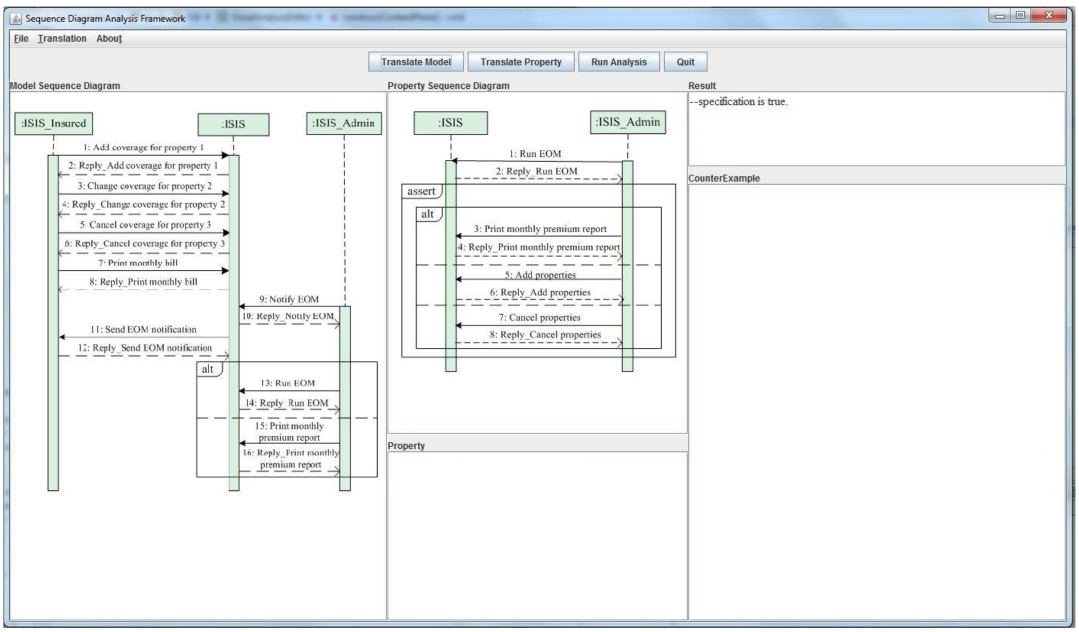


Figure 7.3: Screenshot of the tool suite (Case Study 2).

Chapter 8: EVALUATION

In this section, we validate our technique and tool suite with two case studies. First, we evaluate the tool suite with an insurance industry software application. Second, we evaluate the usability of our technique by modeling HIPAA privacy policies.

8.1 Verify Insurance Software Application Using Tool Suite

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.

8.1.1 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 8.1.

8.1.2 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 contains 3 Lifelines, 16 Messages and a Parallel Combined Fragment with 2 Operands. The Sequence Diagram of this example is shown in figure 8.3.

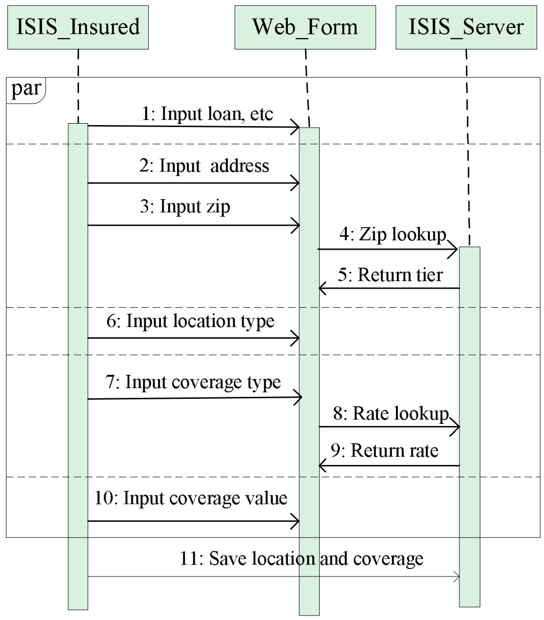


Figure 8.1: Adding coverage to a location

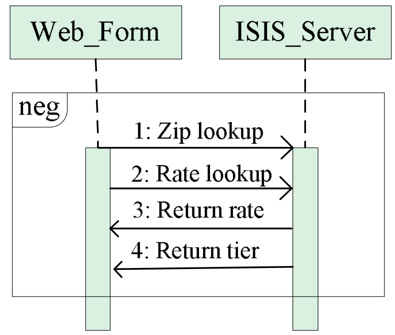


Figure 8.2: Safety property

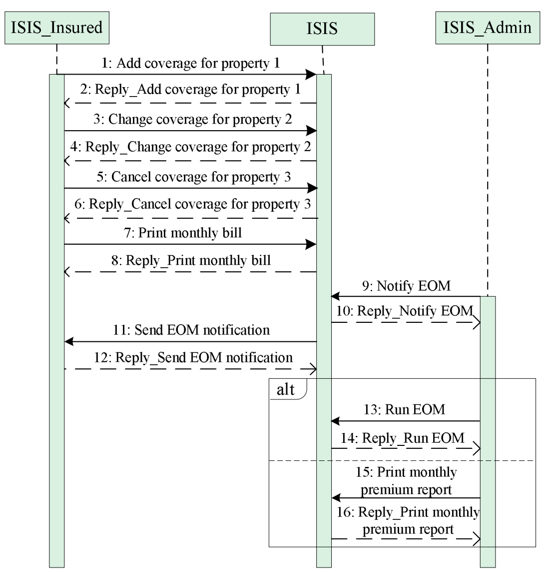


Figure 8.3: ISIS End-of-Month procedure

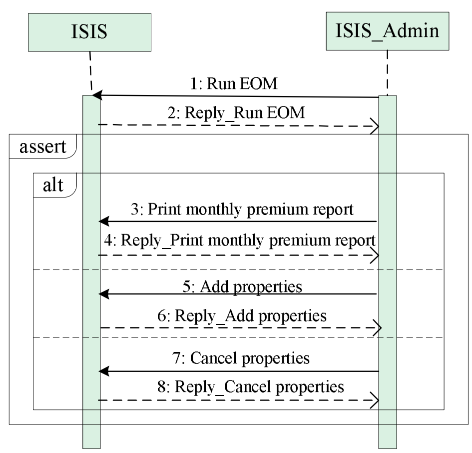


Figure 8.4: Consistency property

8.1.3 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 an NCF shown in figure 8.2). 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 8.5), 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 safety property of a Sequence Diagram with ASCF (see [62] for the diagram) 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.

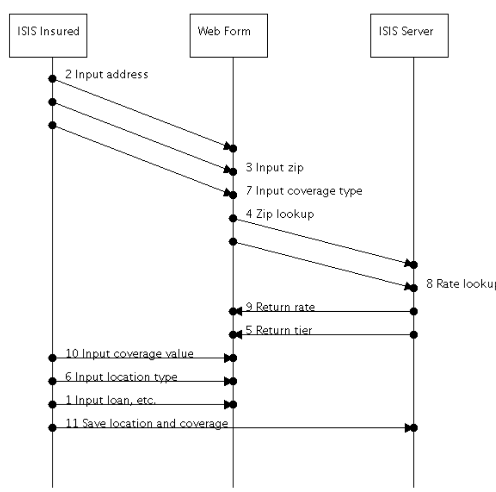


Figure 8.5: Visualization for the counterexample of case study 1

8.2 Modeling HIPAA Policy Using Sequence Diagrams

We evaluate our technique by modeling HIPAA regulations using Sequence Diagrams. Our evaluation focuses on the transmission-related privacy polices.

8.2.1 HIPAA Overview

Nowadays, the widespread use of electronic information, makes the information management for organizations, such hospital, bank, and academic institution, become more convenient. However, the storage and transmission of personal information via networks may cause serious risks. For instance, hackers in Eastern Europe stole personal information on more than $181,000$ people from Department of Technology Services server in Utah [50]. Thus, privacy has become an important concern for organizations to ensure the use and transmission of personal information in compliance with the privacy regulations, such as Payment Card Industry (PCI) Data Security Standard [17] and Health Insurance Portability and Accountability Act of 1996 (HIPAA) [1].

HIPAA provides national standards for insurance portability, fraud enforcement and administrative simplification of the healthcare industry [10]. It regulates the transmission and use of confidential health information, which are referred as protected health information (PHI) among covered entities. Covered entities are the organizations required to comply with HIPAA, including hospitals, insurance companies, doctors and so on. Covered entities who violate HIPAA regulations may face civil and severe criminal penalties [10]. For instance, a former UCLA Health System employee was sentenced to prison and fined for unauthorized access to organizational electronic health record system and to view the medical record [22]. The organizational policies of the covered entities should comply with the HIPAA policies. Failure to comply with HIPAA regulations may cause severe loss. For instance, Rite Aid Corporation paid $1 million for violations of the HIPAA privacy rule, and agreed to improve their policies to safeguard the privacy of their customers [56]. One goal of HIPAA regulations is to prevent the disclosure of PHI of individual to unauthorized people or organizations.

However, as laws are written in legal languages, HIPAA regulations are too complex and dense for policy writers and users of regulated organization to regulate their organizational policies and the transmissions of electronic information. DeYoung et al. have formalized the transmission-related portion of HIPAA privacy rule using a privacy logic, PrivacyLFP, to enforce privacy regulations [20, 21]. Understanding the logical representation is much more complicated and difficult for users without expertise. As a graphical notation, Sequence Diagram is more intuitive and user-friendly. It is deployed to model dynamic behaviors among system actors and their environment through message passing. Thus, Sequence Diagram is an appropriate candidate to model HIPAA regulations. HIPAA regulations consist of general administrative requirement, administrative requirements, security rule, and privacy rule. We are interested in HIPAA privacy rule, which focuses on protecting the privacy of individually identifiable health information during information transmission.

8.2.2 Mapping Strategy

In HIPAA regulations, subpart E of part 164, which defines policies for privacy of individually identifiable health information, consists of 17 sections. We are interested in the sections which are related to information transmission and communication using Sequence Diagrams on the basis of the formal semantics given by our formal framework. Each section contains multiple paragraphs. We consider that each paragraph expresses a privacy policy. We categorize the of HIPAA privacy policies into sufficient policies, necessary policies and exceptional policies. Sufficient policies provide possible means to regulate the behaviors, *i.e.*, each policy may be satisfied if the transmission's purpose meets the policy's purpose. For instance, $\S164.512(a)(1)$ expresses a sufficient policy, which regulates that *"a covered entity may use or disclose protected health information to the extent that such use or disclosure is required by law..."*. Necessary policies provide mandatory means to regulate the behaviors, *i.e.*, each policy must be satisfied if the transmission's purpose meets the policy's purpose. For instance, $\S164.508(a)(2)$ expresses a necessary policy, which regulates that *"a cover entity must obtain an authorization for any use or disclosure of psychotherapy* *notes, except...*". $\S 164.508(a)(2)$ also defines its exceptions, which enumerate the cases that do not need to meet the policy. We consider that these exceptions express exceptional policies, which can be either sufficient policies or necessary policies. To combine the policies, each transmission should satisfy at least one sufficient policy and all the necessary policies. For a policy with exceptions, each transmission should satisfy the policy or one of its exceptional policies. A policy may refer to other policies.

We map each section as a Sequence Diagram with a Parallel CF, where the Parallel contains two Operands. One Operand contains an Alternatives CF for all the sufficient policies, each of which is mapped to an Operand of the Alternatives. The other Operand contains a Parallel CF for all the necessary policies, each of which is mapped to an Operand of the Parallel. Each Operand of a policy refers to a Sequence Diagram illustrating the detail of the policy using the Interaction Use.

We model each policy using a Sequence Diagram with Constraints, where the Constraints represent the purposes of the policy and the predicates in the policy. A predicate represents a condition which needs to be evaluated by external actors. Each actor is modeled using a Lifeline, where the actor's role is modeled using the instance's class. For instance, a Lifeline's head can be $p1 : coveredEntity$, which represents that the role of actor $p1$ is a covered entity. The roles of actors are hierarchial. For a paragraph, the roles in it can be detailed in other paragraphs it refers to. Each message transmitted among actors is modeled using an asynchronous Message, where the message name may indicate the information it carries, *i.e.*, the attributes of an actor. To evaluate the predicates, we add Messages between Lifelines for actors and an external Lifeline representing the evaluator of the predicates. An actor sends a request of a predicate to the evaluator and the evaluator replies the result of the evaluation. If the policy has exceptions, the policy is mapped to a Sequence Diagram with an Alternatives CF, where each exceptional policy is mapped to an Operand of the Alternatives.

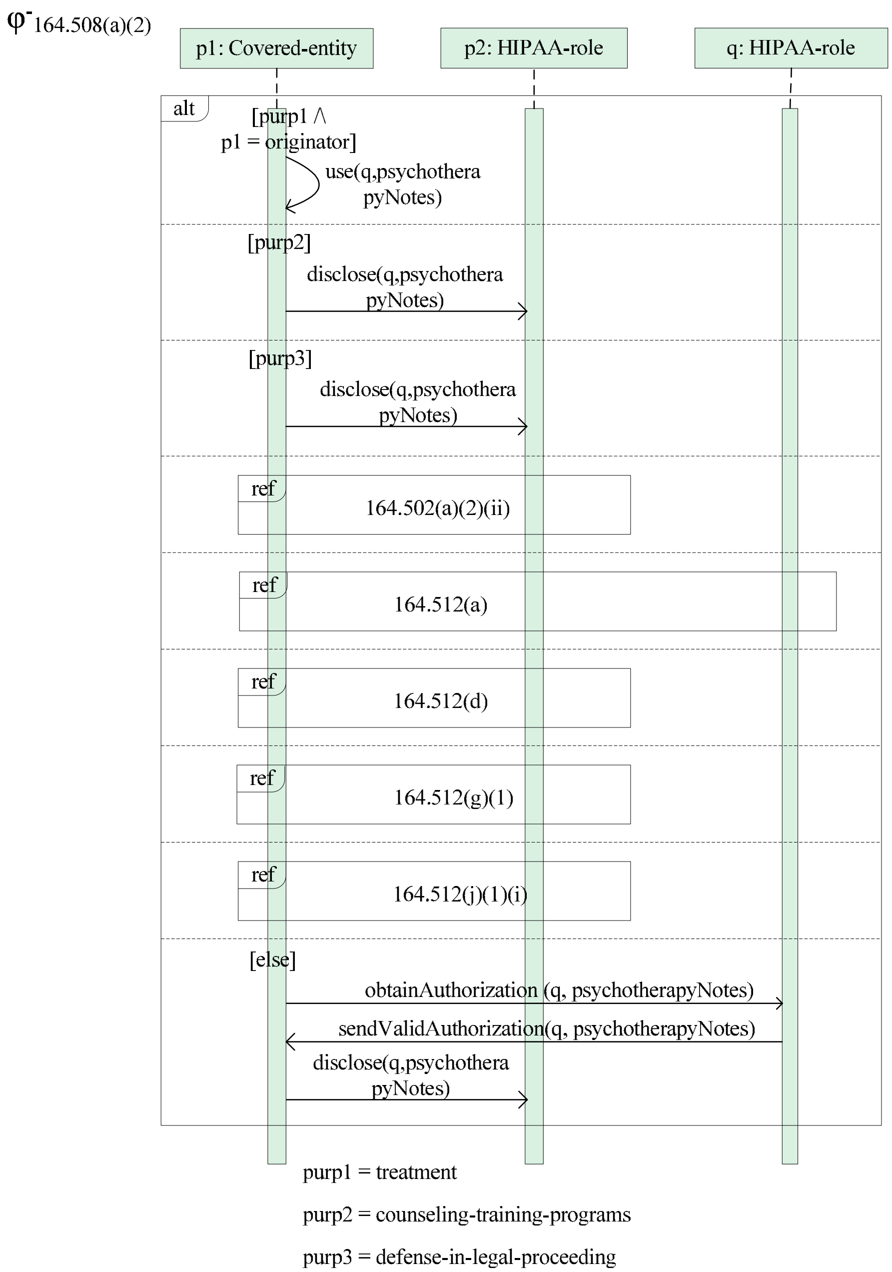


Figure 8.6: Sequence Diagram for paragraph $164.508(a)(2)$

8.2.3 Sequence Diagram Examples for HIPAA Policy

We illustrate our mapping strategies using two examples. Paragraph $164.508(a)(2)$ represents that the use and disclosure of psychotherapy notes requires authorization, and the exceptions. We map the paragraph into a Sequence Diagram with an Alternatives CF (see figure 8.6). The Operand whose Constraint is $else$ expresses the necessary policy, *i.e.*, the authorization is mandatory. $p1$ requests the authorization for use and disclosure of q's psychotherapy notes from q, and q replies with the authorization. Then, $p1$ can disclose to $p2$ q's psychotherapy notes. $p1$'s role is covered-entity, while $p2$ and q's role are not defined explicitly. We use *HIPAA role* to represent the most general role in HIPAA privacy policies, which can be detailed in the paragraphs it refers to. Other Operands of the Alternatives enumerate the exceptions, each of which has its purpose. The first three Operands express the cases for treatment, counseling training programs, or defense in legal proceeding. The rest Operands express the cases using Interaction Uses, each of which refers to another Sequence Diagram. One case refers to paragraph $164.512(a)$, which is mapped into a Sequence Diagram with an Alternatives CF (see figure 8.7). To represent paragraph $164.512(a)$, DeYoung et al. only include the necessary policies defined in paragraphs $164.512(c)$, $164.512(e)$, and $164.512(f)$ [21]. Actually, we consider that paragraph $164.512(a)$ discusses both sufficient and necessary policies in paragraphs $164.512(c)$, $164.512(e)$, and $164.512(f)$. Paragraph $164.512(a)$ defines a sufficient policy, which regulates that a covered entity can use or disclose PHI, including psychotherapy notes, if it is required by law. In the Sequence Diagram, a covered entity, $p1$, requests the external evaluator to evaluate the predicate, *i.e.*, whether the use or disclosure is required by law, and the evaluator replies the result. If the predicate evaluates to $true$, the covered entity should meet the requirements in $164.512(c)$, $164.512(e)$, or $164.512(f)$ to use or disclose the PHI, which is expressed using the Alternatives CF. The paragraphs of $164.512(c)$, $164.512(e)$, and $164.512(f)$ define sufficient policies, at least one of which should be satisfied. Operands of a CF may contain different Lifelines, expressing that different actors are involved in different policies or an actor belongs to different roles in different policies. For instance, q is a victim of abuse in $164.512(c)$, while it is a victim of crime in $164.512(f)$.

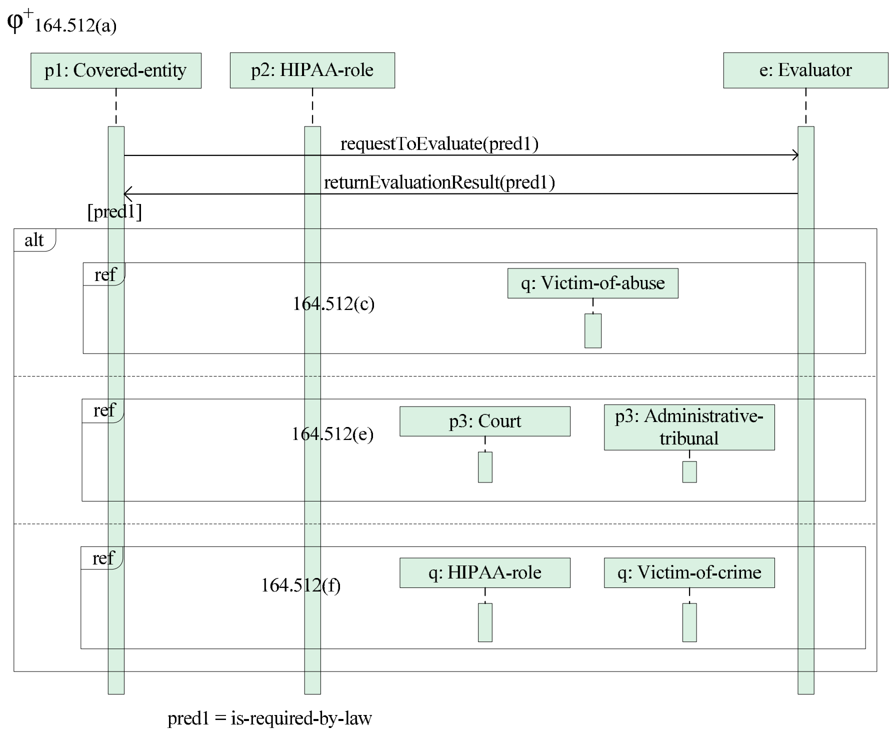


Figure 8.7: Sequence Diagram for paragraph $164.512(a)$

8.2.4 Benefit and Limitation

Modeling HIPAA privacy policies using Sequence Diagrams help user to gain a better understanding of the policies, avoiding the penalty and loss of the violations. The Sequence Diagrams expressing privacy policies can also be translated into logical formulas using our tool suite. The users and organizational policy writers can model their transmissions of electronic health information and organizational policies using Sequence Diagrams, which can be verified against the HIPAA privacy policies with our tool suite. We believe that it helps the the organizational policy writers and users to verify whether their policies or the transmissions of electronic health information comply with HIPAA privacy policies.

Sequence Diagram is used to model the dynamic interaction among actors. Therefore, we cannot model the static, abstract requirements in HIPAA privacy poicies, such as *"a valid autho-*

rization must contain at least the following element.” using Sequence Diagrams.

Chapter 9: RELATED WORK

This chapter provides a literature review of the works in related fields. We start by discussing other formalizations of scenario-based modeling languages. Then, we describe related works which synthesize state-based models from scenario-based models. Finally, we discuss the approaches verifying the scenario-based notations.

9.1 Semantics of Scenario-Based Models

To the best of our knowledge, our technique is the first to support all CFs and the nested CFs. Micskei and Waeselynck survey comprehensively formal semantics proposed for Sequence Diagrams by 13 groups and present the different options taken in [51]. In these groups, [39] 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. Similarly, our approach provides safety and liveness properties. Work towards the similar goal, Cavarra and Filipe propose an approach to express liveness properties using Sequence Diagrams with the concepts from LSC [3]. They also provide a semantics for these Sequence Diagrams using abstract state machines [4]. On the basis of a denotational semantics of Interactions [13], Cengarle and Knapp define an operational semantics of Sequence Diagram [14]. The semantics differentiates positive fragments and negative fragments, concentrating on the overspecialized negative fragments. Eichner et al. introduce a compositional formal semantics of UML 2 Sequence Diagram using colored high-level Petri Nets [27]. The semantics represents a subset of the CFs of Sequence Diagrams. They also deconstruct a Sequence Diagram into fragments, each of which covers multiple Lifelines while each of our fragments covers one Lifeline. Fernandes et al. also formalize UML use case and Sequence Diagram using colored Petri Nets [29]. Their approach only supports several Operators, including Alternatives, Option, Parallel, Loop, and Interaction Use. Filipe provides a formal semantics for several Interaction Operators, including Alternatives, Parallel, Weak Sequencing and Interaction Use [43]. To capture the mandatory and possible behaviors, The semantics adopts the hot and cold Messages from LSC. Hammal defines a denotational semantics, which also formalizes the time constraints of Sequence Diagram [33] . Dan et al. present a trace semantics to express multi-threaded objects using Sequence Diagram, where each Lifeline can capture multiple threads. Only the Messages of the same thread are ordered as their graphical order. Previously, we also provide a formal semantics of Sequence Diagram using template semantics [19]. We support most of the Interaction Operators, except for Ignore and Consider.

Lamsweerde et al. [70] 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 [44] 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 [72]. 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 [73]. The generated hierarchical state machines are used to simulate scenarios and improve readability. Our work focuses on Sequence Diagrams in level-3. Balancing flexibility and simplicity in expressing temporal properties, Mitchell [52] demonstrates that there is a unique minimal generalization of a race-free partial-order scenario even if it is iterative. Mitchell [53] also extends the Mauw and Reniers' algebraic semantics for formalizing the MSC to describe the UML 2 Sequence Diagram, whose deadlock property is defined differently from ours.

9.2 Synthesis of Scenario-Based Models

To analyze multiple scenario-based models of a system, many approaches synthesize state-based models from scenario-based models, where a state-based model is often to represent the behavior of the entire system. Uchitel et al. [68] synthesize a behavioral specification in the form of a Finite Sequential Process, which can be checked using their labeled transition system analyzer. They define the semantics of MSC in terms of labeled transition systems and parallel composition, and translate scenarios into a behavioral specification, which can be analyzed. Working towards similar goals, Damas et al. synthesize a labeled transition system model from both positive and negative scenarios, expressed in MSC [18]. They adopt the semantics definitions from [68]. In addition, they generate temporal properties from scenarios. Whittle and Schumann [74] 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. Similarly, Uchitel et al. synthesize behavior models in terms of Modal Transition System from a combination of safety properties and scenarios [67]. They would like to differentiate the required, possible, and proscribed behavior. Our work formalizes all the CFs in LTL, which helps us to synthesize a collection of Sequence Diagrams.

A comprehensive survey of these synthesis approaches and others' work can be found in [46], where the authors survey 21 synthesis approaches. 7 approaches select Sequence Diagram as the source notation. Makinen and Systa define an interactive algorithm, Minimally Adequate Synthesizer, to synthesize UML statechart from Sequence Diagrams [49]. They check the completeness of the Sequence Diagrams and try to detect the implied scenarios. To evaluate MAS, they implement it and integrate with a software development tool, the Nokia TED. Towards the similar goal, Maier and Zundorf provide automated tool support to derive statecharts from Sequence Diagram [48]. The tool iteratively refines the system, from textual scenario to Java code. Ziadi et al. start to consider the Interaction Operators of UML 2 Sequence Diagram for synthesis. They provide an algebraic framework to synthesize statecharts from Sequence Diagrams with Alternatives, Weak Sequencing, or Loop [75]. Nicolas and Martinez use Sequence Diagram and use case diagram to present the service model [36]. They provide patterns to illustrate the dependencies between Sequence Diagrams using use case diagram. With the dependencies, they synthesize state machine from Sequence Diagram and detect the inconsistencies among Sequence

Diagrams. In order to generates a user interface prototype from scenarios, Elkoutbi and Keller translate Sequence Diagrams into Colored Petri Nets [28]. Similarly, Kloul and Kuster-Filipe translate Sequence Diagram into a process algebra, PEPA [38]. Their approach covers several CFs, including Alternatives, Parallel, and Loop.

9.3 Analysis of Scenario-Based Models

Approaches which formalize scenario-based models or synthesize state-based models from scenario-based models have a common use: analysis. In previous sections, we have listed the approaches which analyze scenario-based models. In addition to these approaches, Alur et al. present a formal semantics of MSCs based on automata theory [7] to model check MSCs. They synthesize state machines from MSCs and detect safe realizability to infer missing scenarios for realizing deadlock-free implementation [6]. They also examine different cases of MSC verification of temporal properties and present techniques for iteratively specifying requirements [5]. They focus on MSC Graphs (an aggregation of MSCs) and techniques for determining if a particular MSC is realized in an MSC Graph. Peled presents an efficient model checking algorithm, which is an extension to SPIN model checking system, for analyzing MSC [57]. They also specify safety and liveness properties of MSC in temporal logic. Our technique can be extend to accommodate their approach as UML 2 Sequence Diagrams have similar expressive features. The provide an algorithm to validate if some execution traces represented using MSC do not consistent with system specification [54]. Their group also extends the MSC standard to represent unmatch Message, intending to model asynchronous Message protocols automatically [32]. Leue et al. translate the MSC specification, especially branching and iteration of High-Level MSC, into PROMELA to verify MSCs using the XSPIN tool [45]. As Sequence Diagrams have similar expressive features, our technique can be extended to work with their approach. 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 [12]. Their work focuses on providing an algorithm and proving the complexity for each problem. Walkinshaw and Bogdanov [71] 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.

Chapter 10: CONCLUSION AND FUTURE WORK

As a well-accepted scenario-based notation, Sequence Diagram is widely used to model the interactions among multiple actors and the environment in reactive system at the requirement and design stages. The lack of formal semantics of Sequence Diagram with CFs makes it difficult to comprehend and analyze the behavior of the system. Model checking, as a common verification technique, automatically and exhaustively enumerates all possible executions of a finite model, and verify whether the executions satisfy desired properties. The gap between graphical notations and input language of model checking tool prevents us to reason about the Sequence Diagrams using verification techniques.

In this dissertation, we present a novel formal framework to formalize the semantics of Sequence Diagrams and all 12 CFs with LTL formulas. To facilitate codifying the semantics of Sequence Diagrams, We deconstruct Sequence Diagrams and CFs to obtain fine-grained syntactic constructs. We provide a collection of simple LTL definitions to represent each semantic aspect of Sequence Diagram as a separate concern. This enable us to conquer the complexity of CFs. The semantic aspects common to Sequence Diagrams and all CFs are captured as a conjunction of the separate simpler LTL definitions. To capture the specific semantic aspect of each CF, we introduce additional constraints, which can be conjuncted with the LTL definition of common semantics. Similarly, the semantics of nested CFs can be captured using conjunctions of LTL definition. To our best knowledge, our formal framework is the first one to support all CFs, the nested CFs, both asynchronous and synchronous Messages, and Interaction Constraints. We also prove that the LTL templates capture the semantic aspects of Sequence Diagram with CFs. We believe our approach can be extended to define the semantics of variants of Sequence Diagram and even other scenario-based languages.

Our approach enables software practitioners to verify if a Sequence Diagram satisfies specified properties and if a set of Sequence Diagrams are consistent. We present a Sequence Diagram with CFs using NuSMV modules. With the help of deconstruction, we codify the semantics of

Sequence Diagrams and CFs in the input language of NuSMV. We also prove that the NuSMV model captures the semantic aspects of Sequence Diagram with CFs. Our approach is also the first to support all CFs and the nested CFs.

One of the key benefit of our formal framework is expressing high-level objectives. We translate the Assertion CFs, which describe the mandatory behaviors, and the Negative CFs, which describe the forbidden behaviors, into LTL formulas to express safety properties. We can check the model against the safety properties without specifying the properties directly. If the properties are not satisfied, counterexamples are visualized as Sequence Diagrams to help practitioners locate violations. We supplement our technique with a proof-of-concept tool suite.

To validate our technique and tool suite, we provide two case studies. First, we perform a case study of an industry web application to evaluate our tool suite. Second, we model the HIPAA privacy policies using Sequence Diagram with CFs. This helps users to gain a better understanding of the HIPAA regulations, avoiding penalty and loss of violations. We believe our technique and tool suite can assist users and organizational policy writers to verify whether the transmissions of electronic health information and organizational policies comply with HIPAA regulations.

Our future work includes two tasks. First, we plan to extend our approach to define the semantics of variants of Sequence Diagrams. We also plan to define the cases discussed in 4.5.3 using our formal framework. Second, we plan to finish modeling all HIPAA privacy policies which are related to information transmissions. To represent the policies specifying universal/existential behaviors, our formal framework may be extended with additional templates. These templates are composed with existing templates to express the semantics of the Sequence Diagram with universal/existential constructs. We intend to verify a model for electronic information transmission or organizational policy using our tool suite to verify whether the model is in compliance with HIPAA privacy policies.

Appendix A: AUXILIARY FUNCTIONS

Our formal 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:

$$AOS(CF) = \begin{cases} TOS(CF) \cup \bigcup\limits_{CF_i \in nested(CF)} AOS(CF_i) & if(typeCF(CF) \neq alt) \quad (1) \\ TOS(m) \cup \bigcup\limits_{CF_i \in nested(CF)} AOS(CF_i) & if(typeCF(CF) = alt) \quad (2) \end{cases}$$

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 $nested(u)$ are introduced to make the templates succinct. For instance, $TBEU(u)$ can be represented as

$$\bigwedge_{g \in TBEU(k\uparrow_i)} \alpha_g = \bigwedge_{h \in ABEU(k\uparrow_i)} ((CND(h) \wedge \alpha_h) \vee (\neg CND(h))).$$

We introduce functions *pre(u)* and *post(u)* to return the set OSs which happen right before or right after CEU u in section 4. The functions *pre(u)* and *post(u)* take the CEU *u* and (by default) the Sequence Diagram as arguments. To calculate the *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 *True*, *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 *True* and contains no nested CEUs, *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 *True* and contains no nested CEU,

  - Case3.1: *v* with Operator "par" obliges *pre(u)* to return a set containing the last OS of each BEU;

  - Case3.2: *v* with Operator"alt" forces *pre(u)* to return a singleton set containing the last OS of the chosen BEU; (We introduce a variable "exe" for BEU of each Operand to indicate the chosen BEU $\widehat{\bigvee_{i\in[1..m]}} exe_i \wedge \bigwedge_{i\in[1..m]} (exe_i \rightarrow cond_i)$, where *m* is the number of BEUs.);

  - Case3.3: *v* with Operator "weak" or "strict" makes *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 *False* or a BEU whose condition evaluates to *False*, we check the BEU or CEU prior to *v* until a BEU or a CEU with at least one EU whose condition evaluates to *True* is found. *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 $True$, we focus on EU q 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, *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 $True$,

- Case5.2.1:*v* with Operator "par" makes we recursively apply case 1 or case 5.1 to each EU whose conditions evaluate to $True$
- Case5.2.2:*v* with Operator"alt" makes we recursively apply case 5.1 to the chosen EU. ($\widehat{\bigvee_{i \in [1..m]}} exe_i \wedge \bigwedge_{i \in [1..m]} (exe_i \rightarrow cond_i)$, where *m* is the number of BEUs.)
- Case5.2.3: *v* with Operator "weak" or "strict" makes we recursively apply case 5.1 to the last EU.

post(u) can be calculated in a similar way.

Appendix B: PROOFS

In this appendix, we provide the proofs for theorem 4.9, theorem 4.11 and theorem 4.12 in chapter 4. We also provide the proofs for theorem 6.14 and theorem 6.16 in chapter 6.

B.1 Proof of Theorem 4.9

Theorem 4.9. For a given Sequence Diagram, seq, with j Messages, $(\Sigma_{sem}^{seq})^*$ and $PRE_{2j}((\Sigma_{LTL}^{seq})^{\omega})$ are equal.

Proof. We use mathematical induction, which is based on the number of Messages, j, within seq.

Base step. Basic Sequence Diagram seq_1 contains only one Message, m_1. ($j = 1$)

- Case 1. Sending OS s_1, and receiving OS r_1 of Message m_1 locate on two Lifelines L_1, L_2 respectively (see figure B.1).

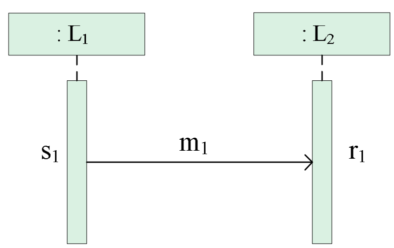


Figure B.1: Case 1 for basic Sequence Diagram with single Message

$\Sigma_{sem}^{seq_1} = \{s_1, r_1\}$, where $\Sigma_{sem}^{seq_1} \subseteq \Sigma$. The semantic aspects of seq_1 define that, for m_1, r_1 can only happen after s_1. Only one trace, $v =< s_1, r_1 >$ of size 2, can be derived from seq_1, *i.e.*, $(\Sigma_{sem}^{seq_1})^* = \{< s_1, r_1 >\}$.

We wish to prove that $< s_1, r_1 > \cdot\tau^\omega \models \widetilde{\Pi}^{Basic}_{seq_1}$, in which $\widetilde{\Pi}^{Basic}_{seq_1}$ for seq_1 is shown as below.

$$\begin{aligned}
\widetilde{\Pi}^{Basic}_{seq_1} =& \tilde{\alpha}_{seq\uparrow_{L_1}} \wedge \rho_{m_1} \wedge \beta_{m_1} \wedge \varepsilon_{seq_1} \\
\rho_{m_1} =& (\neg s_1 \, \widetilde{\mathcal{U}} \, (s_1 \wedge \bigcirc\Box\neg s_1)) \wedge (\neg r_1 \, \widetilde{\mathcal{U}} \, (r_1 \wedge \bigcirc\Box\neg r_1)) \\
\beta_{m_1} =& \neg r_1 \, \widetilde{\mathcal{U}} \, s_1 \\
\varepsilon_{seq_1} =& \Box((\neg s_1 \wedge r_1) \vee (s_1 \wedge \neg r_1) \vee ((\widehat{\Diamond} s_1) \wedge (\widehat{\Diamond} r_1)))
\end{aligned}$$

Sub-formula $\tilde{\alpha}_{seq\uparrow_{L_1}}$ returns $true$ because Lifeline L_1 contains only one OS, s_1. $< s_1, r_1 > \cdot\tau^\omega$ satisfies sub-formula ρ_{m_1} because s_1 and r_1 only occur once. It satisfies sub-formula β_{m_1} because s_1 happens before r_1 does. It also satisfies sub-formula ε_{seq_1} because only one OS happens at a time and $< s_1, r_1 >$ executes uninterrupted. Thus, $< s_1, r_1 > \cdot\tau^\omega \models \widetilde{\Pi}^{Basic}_{seq_1}$.

We wish to prove that $\forall\sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma^{seq_1}_{LTL})^\omega$, then $\sigma_{[1..2]} \in (\Sigma^{seq_1}_{sem})^*$.

σ satisfies sub-formula ρ, which constrains that s_1 and r_1 can occur once and only once respectively. Therefore, $\sigma_{[1..2]}$ can be $< s_1, r_1 >$ or $< r_1, s_1 >$. Sub-formula β_{m_1} represents that r_1 cannot occur until s_1 does. Therefore, $\sigma_{[1..2]}$ can only be $< s_1, r_1 >$, which is an element of $(\Sigma^{seq_1}_{sem})^*$. In this way, we can prove $\sigma_{[1..2]} \in (\Sigma^{seq_1}_{sem})^*$.

- Case 2. Sending OS s_1, and receiving OS r_1 of Message m_1 locate on a single Lifeline L_1 (see figure B.2).

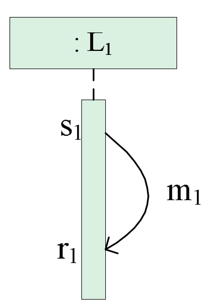


Figure B.2: Case 2 for basic Sequence Diagram with single Message

Besides the semantic aspects discussed in case 1, the OSs on L_1 respect their graphical order, i.e., s_1 occurs before r_1. Trace $\upsilon =< s_1, r_1 >$ of size 2 can be derived from seq_1, i.e., $(\Sigma_{sem}^{seq_1})^* = \{< s_1, r_1 >\}$.

$\widetilde{\Pi}_{seq}^{Basic}$ is reduced to $\widetilde{\Pi}_{seq_1}^{Basic}$ for seq_1 as below.

$$
\begin{aligned}
\widetilde{\Pi}_{seq_1}^{Basic} =& \tilde{\alpha}_{seq_1\uparrow_{L_1}} \wedge \beta_{m_1} \wedge \rho_{m_1} \wedge \varepsilon_{seq_1} \\
\tilde{\alpha}_{seq_1\uparrow_{L_1}} =& \neg r_1 \, \widetilde{\mathcal{U}} \, s_1 \\
\rho_{m_1} =& (\neg s_1 \, \widetilde{\mathcal{U}} \, (s_1 \wedge \bigcirc \Box \neg s_1)) \wedge (\neg r_1 \, \widetilde{\mathcal{U}} \, (r_1 \wedge \bigcirc \Box \neg r_1)) \\
\beta_{m_1} =& \neg r_1 \, \widetilde{\mathcal{U}} \, s_1 \\
\varepsilon_{seq_1} =& \Box ((\neg s_1 \wedge r_1) \vee (s_1 \wedge \neg r_1) \vee ((\widehat{\Diamond} s_1) \wedge (\widehat{\Diamond} r_1)))
\end{aligned}
$$

Comparing to $\widetilde{\Pi}_{seq_1}^{Basic}$ in case 1, only sub-formula $\tilde{\alpha}_{seq_1\uparrow_{L_1}}$ is changed. $\tilde{\alpha}_{seq_1\uparrow_{L_1}}$ represents that s_1 happen before r_1, which enforces the same order as sub-formula β_{m_1}. Trace $< s_1, r_1 > \cdot \tau^\omega$ can be generated from $\widetilde{\Pi}_{seq_1}^{Basic}$, i.e., $(\Sigma_{LTL}^{seq_1})^\omega = \{< s_1, r_1 > \cdot \tau^\omega\}$.

Similarly, we wish to prove that $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_1}^{Basic}$; and $\forall \sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, then $\sigma_{[1..2]} \in (\Sigma_{sem}^{seq_1})^*$. The proof follows the one of case 1.

To sum up, for a basic Sequence Diagram with one Message, $(\Sigma_{sem}^{seq})^*$ and $pre((\Sigma_{LTL}^{seq})^\omega)$ are equal.

Inductive step. Basic Sequence Diagram seq_n contains n Messages, which are graphically-ordered, i.e., (m_{i-1} locates above m_i $(2 \le i \le k)$). The Messages have $2n$ OSs, which locate on k Lifelines. We assume $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_n})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_n}^{Basic}$; and $\forall \sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_n})^\omega$, then $\sigma_{[1..2n]} \in (\Sigma_{sem}^{seq_n})^*$ $(j = n)$.

We add a Message, m_{n+1}, at the bottom of seq_n graphically to form a new Sequence Diagram, seq_{n+1}, with $n+1$ Messages. We wish to prove $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \models \Pi_{seq_{n+1}}^{Basic}$; and $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$ $(j = n+1)$.

(a) We wish to prove $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \models \Pi_{seq_{n+1}}^{Basic}$.

The semantic aspects of seq_{n+1} enforce that only one OS occurs at a time, and each OS happens once and only once. $\Sigma_{sem}^{seq_{n+1}} = \Sigma_{sem}^{seq_n} \cup \{s_{n+1}, r_{n+1}\}$, where $|\Sigma_{sem}^{seq_n}| = 2n$ and $|\Sigma_{sem}^{seq_{n+1}}| = 2n+2$. If $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then υ' is a finite trace of size $2n+2$, which contains OSs in $\Sigma_{sem}^{seq_{n+1}}$. Adding m_{n+1} at the bottom of seq_n does not change the structure of seq_n. Thus, for trace υ', the order of OSs in $\Sigma_{sem}^{seq_n}$ is still preserved. Message m_{n+1} restricts that s_{n+1} must happen before r_{n+1}, i.e., s_{n+1} locates before r_{n+1} in υ'.

$$\begin{aligned}
\widetilde{\Pi}_{seq_n}^{Basic} =& \bigwedge_{\substack{i \in LN(seq_n)\\ g = seq_n \uparrow_i}} \tilde{\alpha}_g \wedge \bigwedge_{j \in MSG(seq_n)} \rho_j \wedge \bigwedge_{j \in MSG(seq_n)} \beta_j \wedge \varepsilon_{seq_n} \\
\tilde{\alpha}_g =& \bigwedge_{k \in [r..(r+|AOS(g)|-2)]} (\neg OS_{k+1} \,\widetilde{\mathcal{U}}\, OS_k) \\
\rho_j =& (\neg SND(j) \,\widetilde{\mathcal{U}}\, (SND(j) \wedge \bigcirc\Box\neg SND(j))) \wedge (\neg RCV(j) \,\widetilde{\mathcal{U}}\, (RCV(j) \wedge \bigcirc\Box\neg RCF(j))) \\
\beta_j =& \neg RCV(j) \,\widetilde{\mathcal{U}}\, SND(j) \\
\varepsilon_{seq_n} =& \Box((\widehat{\bigvee_{OS_m \in AOS(seq_n)}} OS_m) \vee (\bigwedge_{OS_m \in AOS(seq_n)} (\widehat{\Diamond} OS_m)))
\end{aligned}$$

$$\begin{aligned}
\widetilde{\Pi}_{seq_{n+1}}^{Basic} =& \bigwedge_{\substack{i \in LN(seq_{n+1})\\ g = seq_{n+1} \uparrow_i}} \tilde{\alpha}_g \wedge \bigwedge_{j \in MSG(seq_{n+1})} \rho_j \wedge \bigwedge_{j \in MSG(seq_{n+1})} \beta_j \wedge \varepsilon_{seq_{n+1}} \\
=& (\bigwedge_{\substack{i \in LN(seq_n)\\ g = seq_n \uparrow_i}} \tilde{\alpha}_g \wedge \varsigma_{seq_n, m_{n+1}}) \wedge (\bigwedge_{j \in MSG(seq_n)} \rho_j \wedge \rho_{m_{n+1}}) \wedge (\bigwedge_{j \in MSG(seq_n)} \beta_j \wedge \beta_{m_{n+1}}) \wedge \varepsilon_{seq_{n+1}} \\
=& (\bigwedge_{\substack{i \in LN(seq_n)\\ g = seq_n \uparrow_i}} \tilde{\alpha}_g \wedge \bigwedge_{j \in MSG(seq_n)} \rho_j \wedge \bigwedge_{j \in MSG(seq_n)} \beta_j) \wedge (\rho_{m_{n+1}} \wedge \beta_{m_{n+1}}) \wedge \varsigma_{seq_n, m_{n+1}} \wedge \varepsilon_{seq_{n+1}} \\
=& \iota_{seq_n} \wedge \vartheta_{m_{n+1}} \wedge \varsigma_{seq_n, m_{n+1}} \wedge \varepsilon_{seq_{n+1}}
\end{aligned}$$

$$\begin{aligned}
\iota_{seq_n} =& \bigwedge_{\substack{i \in LN(seq_n)\\ g = seq_n \uparrow_i}} \tilde{\alpha}_g \wedge \bigwedge_{j \in MSG(seq_n)} \rho_j \wedge \bigwedge_{j \in MSG(seq_n)} \beta_j \\
\vartheta_{m_{n+1}} =& \rho_{m_{n+1}} \wedge \beta_{m_{n+1}} \\
\varepsilon_{seq_{n+1}} =& \Box((\widehat{\bigvee_{OS_m \in AOS(seq_{n+1})}} OS_m) \vee (\bigwedge_{OS_m \in AOS(seq_{n+1})} (\widehat{\Diamond} OS_m)))
\end{aligned}$$

Figure B.3: LTL formulas for seq_n and seq_{n+1}

$\widetilde{\Pi}_{seq}^{Basic}$ is reduced to $\widetilde{\Pi}_{seq_n}^{Basic}$ and $\widetilde{\Pi}_{seq_{n+1}}^{Basic}$ for seq_n and seq_{n+1} respectively (see figure B.3). We group the sub-formulas of $\widetilde{\Pi}_{seq_{n+1}}^{Basic}$ using ι_{seq_n}, $\vartheta_{m_{n+1}}$, $\varsigma_{seq_n, m_{n+1}}$, and $\varepsilon_{seq_{n+1}}$. In order to prove

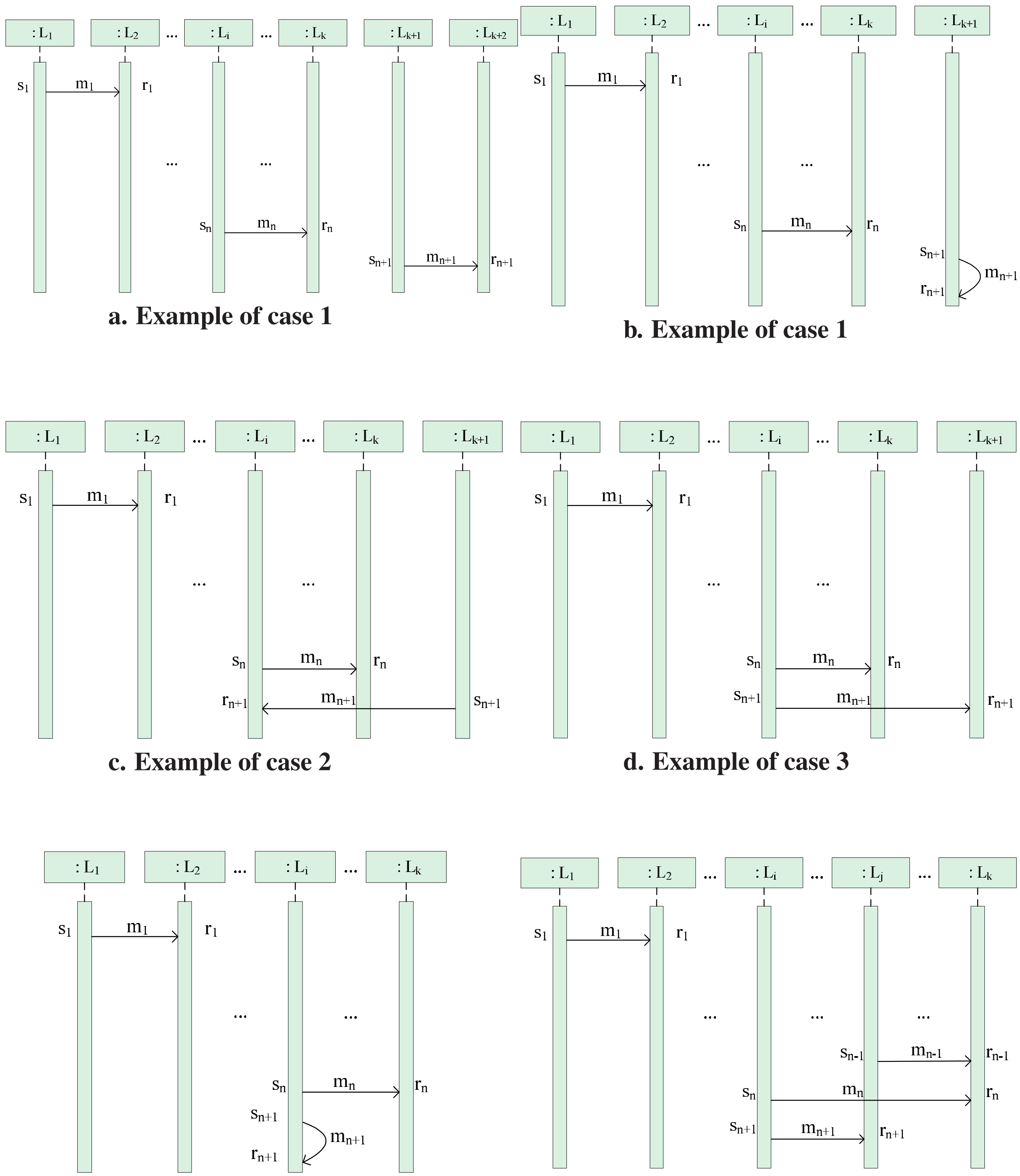
$\upsilon' \cdot \tau^{\omega} \models \widetilde{\Pi}^{Basic}_{seq_{n+1}}$, we wish to prove that $\upsilon' \cdot \tau^{\omega}$ satisfies all sub-formulas of $\widetilde{\Pi}^{Basic}_{seq_{n+1}}$. Sub-formula ι_{seq_n} enforces the order of OSs within seq_n, which includes the order of OSs along each Lifeline, and the order between OSs of each Message. We assume that if $\upsilon \in (\Sigma^{seq_n}_{sem})^*$, then $\upsilon \cdot \tau^{\omega} \models \Pi^{Basic}_{seq_n}$. It is easy to observe that $\upsilon \cdot \tau^{\omega}$ also satisfies ι_{seq_n}. As we discussed, the order of OSs within seq_n is still preserved in υ'. Thus, $\upsilon' \cdot \tau^{\omega}$ satisfies ι_{seq_n}. Sub-formula $\vartheta_{m_{n+1}}$ enforces the order between OSs of m_{n+1}, i.e., s_{n+1} and r_{n+1} happen only once respectively, and s_{n+1} must occur before r_{n+1}. $\upsilon' \cdot \tau^{\omega}$ satisfies $\vartheta_{m_{n+1}}$ because (1) only one s_{n+1} and one r_{n+1} are in υ', and (2) s_{n+1} locates before r_{n+1} in υ'. Sub-formula $\varepsilon_{seq_{n+1}}$ enforces that only one OS of seq_{n+1} can execute at once, and the trace should execute uninterrupted. As we discussed, in $\upsilon' \cdot \tau^{\omega}$, each OS of seq_{n+1} only executes once, and the execution of υ' does not interleaved by τ. Therefore, $\upsilon' \cdot \tau^{\omega}$ satisfies $\varepsilon_{seq_{n+1}}$. $\varsigma_{seq_n, m_{n+1}}$ enforces the order between the OSs of seq_n and the OSs of m_{n+1}. We wish to prove that $\upsilon' \cdot \tau^{\omega}$ satisfies $\varsigma_{seq_n, m_{n+1}}$ using four cases as below.

- Case 1: Two OSs of m_{n+1} locate on two new Lifelines, L_{k+1} and L_{k+2} (see figure B.4a); or two OSs of m_{n+1} locate on one new Lifeline, L_{k+1} (see figure B.4b).

  The OSs of m_{n+1} locate on one or two new Lifelines, so m_{n+1} and the existing Messages, $m_1, m_2 ... m_n$, are interleaved. Therefore, in trace $\upsilon' \in (\Sigma^{seq_{n+1}}_{sem})^*$, s_{n+1} or r_{n+1} can locate (1) between any two OSs of seq_n, or (2) before all OSs of seq_n, or (3) after all OSs of seq_n. Thus, s_{n+1} can be the sth OS of υ', where $1 \leq s \leq 2n+1$; and r_{n+1} can be the rth OS of υ', where $s < r \leq 2n+2$.

  $$\varsigma_{seq_n, m_{n+1}} = \tilde{\alpha}_{seq\uparrow_{L_{k+1}}} \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+2}}}$$

  Sub-formula $\varsigma_{seq_n, m_{n+1}}$ is a conjunction of $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ and $\tilde{\alpha}_{seq\uparrow_{L_{k+2}}}$. Only one OS locates on Lifeline L_{k+1}. Therefore, $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ returns $true$ as defined by sub-formula $\tilde{\alpha}_g$. Similarly, $\tilde{\alpha}_{seq\uparrow_{L_{k+2}}}$ returns $true$. Thus, $\varsigma_{seq_n, m_{n+1}}$ returns $true$. $\upsilon' \cdot \tau^{\omega}$ satisfies $\varsigma_{seq_n, m_{n+1}}$, i.e., $\upsilon' \cdot \tau^{\omega} \models \varsigma_{seq_n, m_{n+1}}$.



a. Example of case 1

b. Example of case 1

c. Example of case 2

d. Example of case 3

e. Example of case 3

f. Example of case 4

Figure B.4: Examples for basic Sequence Diagram with $n+1$ Messages

- Case 2: Sending OS s_{n+1} locates on a new Lifeline, L_{k+1}, and receiving OS r_{n+1} locates on an existing Lifeline, L_i $(1 \leq i \leq k)$ (see figure B.4c).

  In seq_n, we assume the last OS on L_i is OS_{pre}. After adding m_{n+1} at the bottom of seq_n , r_{n+1} becomes the last OS on L_i. Therefore, OS_{pre} should happen before r_{n+1}. s_{n+1} locates on a new Lifeline, so it is interleaved with the OSs of seq_n. However, s_{n+1} must happen before r_{n+1}. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre} is the pth OS, where $1 \leq p \leq 2n+1$. Then s_{n+1} is the sth OS of υ', where $1 \leq s \leq 2n+1$ and $s \neq p$; r_{n+1} is the rth OS of υ', where $s < r \leq 2n+2$ and $p < r \leq 2n+2$.

  $$\varsigma_{seq_n, m_{n+1}} = (\neg r_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre}) \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$$

  Sub-formula $\varsigma_{seq_n, m_{n+1}}$ defines that r_{n+1} does not happen until OS_{pre} does. $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ returns $true$ because only one OS locates on Lifeline $k+1$. In $\upsilon' \cdot \tau^{\omega}$, OS_{pre} locates before OS_{r+1}, i.e., $p < r$. Thus, $\upsilon' \cdot \tau^{\omega}$ satisfies $\varsigma_{seq_n, m_{n+1}}$, i.e., $\upsilon' \cdot \tau^{\omega} \models \varsigma_{seq_n, m_{n+1}}$.

- Case 3: Sending OS s_{n+1} locates on an existing Lifeline, L_i $(1 \leq i \leq k)$, and receiving OS r_{n+1} locates on a new Lifeline, L_{k+1} (see figure B.4d); or two OSs of m_{n+1} locate on an existing Lifeline L_i $(1 \leq i \leq k)$ (see figure B.4e).

  Similarly, we assume the last OS on L_i in seq_n is OS_{pre}. In seq_{n+1}, if s_{n+1} locates on L_i, OS_{pre} should happen before s_{n+1} because OS_{pre} locates above s_{n+1} graphically. For m_{n+1}, r_{n+1} must happen after s_{n+1}. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre} is the pth OS, where $1 \leq p \leq 2n$. Then s_{n+1} is the sth OS of υ', where $p < s \leq 2n+1$; r_{n+1} is the rth OS of υ', where $s < r \leq 2n+2$.

  $$\varsigma_{seq_n, m_{n+1}} = (\neg s_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre}) \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$$

  Sub-formula $\varsigma_{seq_n, m_{n+1}}$ defines that s_{n+1} cannot happen before OS_{pre}. Only one or none OS locates on Lifeline $k+1$, so $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ returns $true$. In $\upsilon' \cdot \tau^{\omega}$, s_{n+1} locates after OS_{pre}, i.e.,

$p < s$. Therefore, $\upsilon' \cdot \tau^{\omega}$ satisfies $\varsigma_{seq_n, m_{n+1}}$, i.e., $\upsilon' \cdot \tau^{\omega} \models \varsigma_{seq_n, m_{n+1}}$.

- Case 4: Two OSs of m_{n+1} locate on two existing Lifelines. Without loss of generality, we assume that sending OS s_{n+1} locates on Lifeline L_i $(1 \le i \le k)$, receiving OS r_{n+1} locates on Lifeline L_j $(1 \le j \le k)$ (see figure B.4f).

  In seq, we assume the last OS on L_i is OS_{pre_s}, and the last OS on L_j is OS_{pre_r}. After adding m_{n+1} at the bottom of seq_n , s_{n+1} becomes the last OS of L_i, and r_{n+1} becomes the last OS of L_j. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre_s} is the p_sth OS, where $1 \le p_s \le 2n$, and OS_{pre_r} is the p_rth OS, where $1 \le p_r \le 2n+1$. Then s_{n+1} is the sth OS of υ', where $p_s < s \le 2n+1$; r_{n+1} is the rth OS of υ', where $p_r < r \le 2n+2$.

  $$\varsigma_{seq_n, m_{n+1}} = (\neg s_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_s}) \wedge (\neg r_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_r})$$

  The first conjunct of sub-formula $\varsigma_{seq_n, m_{n+1}}$ defines that s_{n+1} cannot happen until OS_{pre_s} executes. In $\upsilon' \cdot \tau^{\omega}$, OS_{pre_s} locates before s_{n+1}, i.e., $p_s < s$. Therefore, $\upsilon' \cdot \tau^{\omega}$ satisfies $\neg s_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_s}$. Similarly, we can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies $\neg r_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_r}$. Thus, $\upsilon' \cdot \tau^{\omega}$ satisfies $\varsigma_{seq_n, m_{n+1}}$, i.e., $\upsilon' \cdot \tau^{\omega} \models \varsigma_{seq_n, m_{n+1}}$.

Now we have proven that for all cases, $\upsilon' \cdot \tau^{\omega} \models \varsigma_{seq_n, m_{n+1}}$.

To conclude, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^{\omega} \models \Pi_{seq_{n+1}}^{Basic}$.

(b) We wish to prove $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^{\omega}$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

If $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^{\omega}$, we wish to prove that $\sigma'_{[1..2n+2]}$ respects all the semantic aspects of seq_{n+1}. For $\widetilde{\Pi}_{seq_{n+1}}^{Basic}$, we still group the sub-formulas using ι_{seq_n}, $\vartheta_{m_{n+1}}$, $\varsigma_{seq_n, m_{n+1}}$, and $\varepsilon_{seq_{n+1}}$, *i.e.*,

$$\widetilde{\Pi}_{seq_{n+1}}^{Basic} = \iota_{seq_n} \wedge \vartheta_{m_{n+1}} \wedge \varsigma_{seq_n, m_{n+1}} \wedge \varepsilon_{seq_{n+1}}$$

We assume that if $\sigma \in (\Sigma_{LTL}^{seq_n})^{\omega}$, then $\sigma_{[1..2n]} \in (\Sigma_{sem}^{seq_n})^*$. It is easy to infer that σ satisfies ι_{seq_n}. Sub-formula ι_{seq_n} enforces the order of OSs in $\Sigma_{sem}^{seq_n}$ and each OS should execute once and only once. We can also infer that trace σ' satisfies ι_{seq_n} from $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^{\omega}$. If σ' does not contain an OS in $\Sigma_{sem}^{seq_n}$, then σ' does not satisfies ι_{seq_n}, which defines that each OS in $\Sigma_{sem}^{seq_n}$ should happen once. Therefore, all OSs in $\Sigma_{LTL}^{seq_n}$ executes once and only once in $\sigma'_{[1..2n+2]}$. We wish to prove that, in $\sigma'_{[1..2n+2]}$, all OSs in $\Sigma_{LTL}^{seq_n}$ respect their order defined by semantic aspects of seq_n. $\exists OS_p, OS_q . OS_p, OS_q \in \Sigma_{LTL}^{seq_n}$, the semantic aspects of seq_n define that OS_p must happen before OS_q. In $\sigma'_{[1..2n+2]}$, we assume that the OSs do not respect the same order, *i.e.*, OS_p occurs after OS_q. ι_{seq_n} codifies the semantic aspects of seq_n, so it constraints that OS_p should take place before OS_q. To satisfy ι_{seq_n}, OS_p must occur before OS_q in σ', which contradicts our assumption. Therefore, in $\sigma'_{[1..2n+2]}$, the OSs in $\Sigma_{sem}^{seq_n}$ respect the order defined by semantic aspects of seq_n, i.e., if we remove the OSs not in $\Sigma_{sem}^{seq_n}$ from $\sigma'_{[1..2n+2]}$ to obtain a new trace $\sigma''_{[1..2n]}$, then $\sigma''_{[1..2n]} \in (\Sigma_{sem}^{seq_n})^*$.

Sub-formula $\vartheta_{m_{n+1}}$ specifies that s_{n+1} must occur before r_{n+1}, and both OSs can occur only once. s_{n+1} and r_{n+1} may not locate on the same Lifeline. Thus, $\vartheta_{m_{n+1}}$ codifies the semantics of Message m_{n+1} in seq_{n+1}. In $\sigma'_{[1..2n+2]}$, s_{n+1} and r_{n+1} represent the semantics of m_{n+1}. We have proven each OSs in $\Sigma_{sem}^{seq_n}$ should happen once and only once in $\sigma'_{[1..2n+2]}$, where $|\Sigma_{sem}^{seq_n}| = 2n$, and both of s_{n+1} and r_{n+1} occur only once. Thus, we can deduct that $\varepsilon_{seq_{n+1}}$ captures the semantics, which defines only one OS executing at a time and the $\sigma'_{[1..2n+2]}$ should execute uninterrupted. Now we wish to prove that sub-formula $\varsigma_{seq_n,m_{n+1}}$ codifies the order between the OSs within $\Sigma_{sem}^{seq_n}$ and the OSs of m_{n+1}, which is discussed using four cases as below.

- Case 1: Two OSs of m_{n+1} locate on two new Lifelines, L_{k+1} and L_{k+2} (see figure B.4a); or two OSs of m_{n+1} locate on one new Lifeline, L_{k+1} (see figure B.4b).

$$\varsigma_{seq_n,m_{n+1}} = \tilde{\alpha}_{seq\uparrow_{L_{k+1}}} \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+2}}}$$

  Sub-formula $\varsigma_{seq_n,m_{n+1}}$ is a conjunction of $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ and $\tilde{\alpha}_{seq\uparrow_{L_{k+2}}}$. It returns $true$ only if none or at most one OS locates on each Lifeline. Therefore only one OS locates on L_{k+1} and L_{k+2} respectively. $\varsigma_{seq_n,m_{n+1}}$ represents that m_{n+1} and the Messages of seq_n are interleaved.

No specific order is defined between the OSs of seq_n and the OSs of m_{n+1}. Thus, $\varsigma_{seq_n,m_{n+1}}$ codifies the order between the OSs of seq_n and the OSs of m_{n+1} in seq_{n+1}. In $\sigma'_{[1..2n+2]}$, the OSs of seq_n and the OSs of m_{n+1} respect the order defined by the semantic aspects of seq_{n+1}.

- Case 2: Sending OS s_{n+1} locates on a new Lifeline, L_{k+1} receiving OS r_{n+1} locates on an existing Lifeline, L_i $(i \leq k)$ (see figure B.4c).

$$\varsigma_{seq_n,m_{n+1}} = (\neg r_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre}) \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$$

Sub-formula $\varsigma_{seq_n,m_{n+1}}$ defines that r_{n+1} cannot happen until OS_{pre} executes, where OS_{pre} is the OS which occurs right before r_{n+1} on Lifeline L_i. As the semantic aspect of seq_{n+1} defined, r_{n+1} should locate right below OS_{pre} on Lifeline L_i and OSs execute in their graphical order. $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ returns $true$. It denotes that only s_{n+1} locates on L_{k+1}. Thus, $\varsigma_{seq_n,m_{n+1}}$ codifies the order between the OSs of seq_n and the OSs of m_{n+1} in seq_{n+1}. In $\sigma'_{[1..2n+2]}$, the OSs of seq_n and the OSs of m_{n+1} respect the order defined by the semantic aspects of seq_{n+1}.

- Case 3: Sending OS s_{n+1} locates on an existing Lifeline, L_i $(i \leq k)$, and receiving OS r_{n+1} locates on a new Lifeline, L_{k+1} (see figure B.4d). or two OSs of m_{n+1} locate on an existing Lifeline L_i $(i \leq k)$ (see figure B.4e).

$$\varsigma_{seq_n,m_{n+1}} = (\neg s_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre}) \wedge \tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$$

Similarly, sub-formula $\varsigma_{seq_n,m_{n+1}}$ defines that s_{n+1} cannot happen until OS_{pre} executes, where OS_{pre} is the OS which occurs right before s_{n+1} on Lifeline L_i. As the semantic aspect of seq_{n+1} defined, s_{n+1} should locate right below OS_{pre} on Lifeline L_i and OSs execute in their graphical order. $\tilde{\alpha}_{seq\uparrow_{L_{k+1}}}$ returns $true$. It denotes that none or only one OS locates on L_{k+1}. Therefore r_{n+1} may locate on L_{k+1} or below s_{n+1} on L_i. Thus, $\varsigma_{seq_n,m_{n+1}}$

codifies the order between the OSs of seq_n and the OSs of m_{n+1} in seq_{n+1}. In $\sigma'_{[1..2n+2]}$, the OSs of seq_n and the OSs of m_{n+1} respect the order defined by the semantic aspects of seq_{n+1}.

- Case 4: Two OSs of m_{n+1} locate on two existing Lifelines. Without loss of generality, we assume that sending OS s_{n+1} locates on Lifeline L_i $(i \leq k)$, receiving OS r_{n+1} locates on Lifeline L_j $(j \leq k)$ (see figure B.4f).

$$\varsigma_{seq_n, m_{n+1}} = (\neg s_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_s}) \wedge (\neg r_{n+1} \, \widetilde{\mathcal{U}} \, OS_{pre_r})$$

  Sub-formula $\varsigma_{seq_n, m_{n+1}}$ defines that s_{n+1} cannot happen until OS_{pre_s} has taken place, where OS_{pre_s} is the OS occurring right before s_{n+1} on Lifeline L_i, and r_{n+1} cannot happen until OS_{pre_r} has taken place, where OS_{pre_r} is the OS occurring right before r_{n+1} on Lifeline L_j. As the semantic aspect of seq_{n+1} defined, s_{n+1} should locate right below OS_{pre_s} on Lifeline L_i, and r_{n+1} should locate right below OS_{pre_r} on Lifeline L_j. OSs execute in their graphical order along each Lifeline. Thus, $\varsigma_{seq_n, m_{n+1}}$ codifies the order between the OSs of seq_n and the OSs of m_{n+1} in seq_{n+1}. In $\sigma'_{[1..2n+2]}$, the OSs of seq_n and the OSs of m_{n+1} respect the order defined by the semantic aspects of seq_{n+1}.

Now we have proven that $\sigma'_{[1..2n+2]}$ respects all the semantic aspects of seq_{n+1}, i.e., $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

To conclude, $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$. □

B.2 Proof of Theorem 4.11 and Theorem 4.12

Theorem 4.11. $(\Sigma_{sem}^{seq_r})^*$ and $PRE_{2h+2p}((\Sigma_{LTL}^{seq_r})^\omega)$ are equal.

Proof. We use mathematical induction, which is based on the number of CFs, r, directly enclosed in seq_r.

Base step. The sequence Diagram contains at most one CF, cf_1. $(r \leq 1)$

- Case 1. Sequence Diagram seq_0 contains no CF. $(r = 0)$

  The proof follows the one for basic Sequence Diagram.

- Case 2. Sequence Diagram seq_1 contains only one CF, cf_1. $(r = 1)$

$$\widetilde{\Pi}_{seq_1} = (\bigwedge_{\substack{i \in LN(seq_1) \\ g \in TBEU(seq\uparrow_i)}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq_1)} \rho_j) \wedge (\bigwedge_{j \in MSG(seq_1)} \beta_j) \wedge \Phi^{cf_1} \wedge \varepsilon_{seq_1}$$

  – Case 2.1 We assume that cf_1 has a Operands whose Interaction Constraint evaluate to *False*. The bth Operand contains q_b Messages, where $1 \leq b \leq a$.

$$\Phi^{cf_1} = \eta^{cf_1} = \bigwedge_{i \in LN(cf_1)} ((\bigwedge_{OS_{post} \in post(cf_1\uparrow_i)} (\neg OS_{post})) \, \widetilde{\mathcal{U}} \, (\bigwedge_{OS_{pre} \in pre(cf_1\uparrow_i)} (\diamondsuit OS_{pre})))$$

  (a) We wish to prove that, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

  First, we consider the semantic aspects of the OSs directly enclosed in seq_1. We project seq_1 onto each of its covered Lifelines to obtain a EU. We also project cf_1 onto each of its covered Lifeline to obtain a CEU. Therefore, each EU of seq_1 may contain a CEU of cf_1 and BEUs grouped by the OSs directly enclosed in the EU. Similar to the semantics of an BEU within a basic Sequence Diagram, the semantics of any BEU directly enclosed in the EU of seq_1 specifies that OSs are ordered as their graphical order. If $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, we can easily infer that $\upsilon \cdot \tau^\omega \models \bigwedge_{\substack{i \in LN(seq_1) \\ g \in TBEU(seq_1\uparrow_i)}} \tilde{\alpha}_g$. The semantics of each Message directly enclosed in seq_1 specifies that its receiving OS cannot happen before the sending OS, and both OS can occur once only once. Accordingly, we can easily infer that $\upsilon \cdot \tau^\omega \models \bigwedge_{j \in MSG(seq_1)} \rho_j$, and $\upsilon \cdot \tau^\omega \models \bigwedge_{j \in MSG(seq_1)} \beta_j$.

Then, we consider the semantics of cf_1. It defines that cf_1 does not execute when the Constraints of all the Operands evaluate to *False*. cf_1's preceding Interaction Fragments and succeeding Interaction Fragments are ordered by Weak Sequencing. In this case, cf_1's preceding OS must happen before its succeeding OS on each Lifeline. We use LTL formula η^{cf_1} to capture cf_1's semantics. η^{cf_1} does not specify the order of OSs within Operands because the Operands whose Constraints evaluate to *False* are excluded. We assume that if $\upsilon \cdot \tau^{\omega}$ does not satisfy η^{cf_1}, then η^{cf_1} specifies that, on Lifeline i, cf_1's preceding OS, OS_{pre}, occurs after cf_1's succeeding OS, OS_{post}. However, η^{cf_1} specifies that, on each Lifeline covered by cf_1, its succeeding OS cannot happen until its preceding OS finishes execution. Functions $pre(cf_1 \uparrow_i)$ and $post(cf_1 \uparrow_i)$ return the set of OSs which may happen right before and after CEU $cf_1 \uparrow_i$. In this case, each set contains at most one OS. Thus, OS_{pre} must happen before OS_{post}, which contradicts our assumption. In this way, we can prove that $\upsilon \cdot \tau^{\omega} \models \eta^{cf_1}$.

Finally, we consider the interleaving semantics of seq_1. No OS in cf_1 can executes, so only the OSs directly enclosed in seq_1 can be enabled to execute. We can prove that $\upsilon \cdot \tau^{\omega} \models \varepsilon_{seq_1}$. The proof follows the one for basic Sequence Diagram.

Now we have proven that if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_1}$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, $\sigma_{[1..2h]} \in (\Sigma_{sem}^{seq_1})^*$.

In $\Sigma_{LTL}^{seq_1}$, no OS within cf_1 is enabled to execute because all the Constraints of cf_1's Operand evaluate to *False*. If $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, then $\sigma = \sigma_{[1..2h]} \cdot \tau^{\omega}$, which follows Lemma 4.10. We wish to prove that $\sigma_{[1..2h]}$ respects all the semantics of seq_1. $\sigma \models \widetilde{\Pi}_{seq_1}$, so σ satisfies all sub-formulas of $\widetilde{\Pi}_{seq_1}$. We prove that the sub-formulas capture the semantic aspects as below.

First, we discuss the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j for seq_1. Function $TBEU(seq_1 \uparrow_i)$ returns the BEUs directly enclosed in seq_1 on Lifeline i. These BEUs, which are separated using CEU of cf_1 on Lifeline i, are formed by the OSs directly enclosed in

seq_1. Function $MSG(cf_1)$ returns the set of Messages directly enclosed in cf_1. We can prove that these sub-formulas capture the semantics of OSs directly enclosed in seq_1. The proof follows the one for OSs within basic Sequence Diagram.

Next, we discuss the sub-formula η^{cf_1}. It defines that, on Lifeline i, OSs in $post(cf_1 \uparrow_i)$ cannot happen until OSs in $pre(cf_1 \uparrow_i)$ finish execution. We assume that, if η^{cf_1} does not capture the semantics of cf_1, then on a Lifeline, i, the preceding OS of cf_1, OS_{pre}, happens after the succeeding OS of cf_1, OS_{post}. However, the semantics of η^{cf_1} defines the Weak Sequencing between cf_1's preceding OSs and succeeding OSs, *i.e.*, its preceding OSs must happen before its succeeding OS on the same Lifeline. Therefore, OS_{pre} must happen before OS_{post}, which contradicts our assumption. In this way, we can prove that η^{cf_1} captures the semantics of cf_1.

Finally, we discuss the sub-formula ε_{seq_1}. It represents that only one OS in $|AOS(seq_1)|$ execute at a time, or all OSs in $|AOS(seq_1)|$ have executed. In this case, function $|AOS(seq_1)|$ returns the set of OSs directly enclosed in seq. We can prove that ε_{seq_1} captures the interleaving semantics of seq_1 by following the proof for basic Sequence Diagram.

Now we have proven that $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, it respects all the semantic aspects of seq_1, *i.e.*, $\sigma_{[1..2h]} \in (\Sigma_{sem}^{seq_1})^{*}$.

To conclude, $\forall \upsilon.\upsilon \in \Sigma^{*}$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^{*}$, then $\upsilon \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_1}$, and $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, then $\sigma_{[1..2h]} \in (\Sigma_{sem}^{seq_1})^{*}$.

– Case 2.2 We assume that cf_1 has at least one Operand whose Constraint evaluates to *True*. The Operator of cf_1 is not "alt" or "loop".

$$\Phi^{cf_1} = \Psi^{cf_1} = \tilde{\theta}^{cf_1} \wedge \bigwedge_{i \in LN(cf_1)} \tilde{\gamma}_i^{cf_1} \wedge \varrho^{cf_1}$$

* Case 2.2.1 We assume that, cf_1 has two Operands. One Operand contains p Messages, and its Interaction Constraint evaluates to *True*. The other Operand contains q Messages, and its Interaction Constraint evaluate to *False*. (see figure B.5, where $cond1$ evaluate to *True*, and $cond2$ evaluates to *False*).

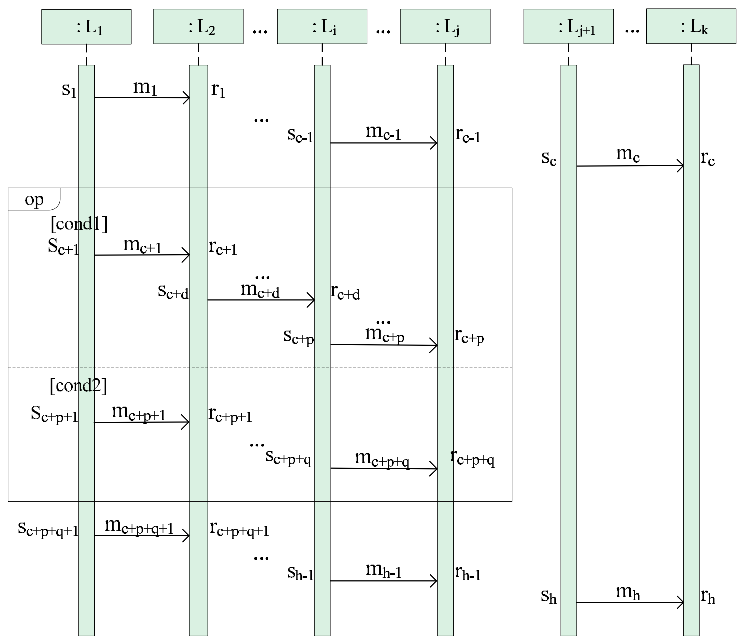


Figure B.5: Example of Sequence Diagram with CF

(a) We wish to prove that, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

First, we consider the semantic aspects of the OSs directly enclosed in seq_1. We can prove that $\upsilon \cdot \tau^\omega$ satisfies $\bigwedge_{\substack{i \in LN(seq_1) \\ g \in TBEU(seq_1\uparrow_i)}} \tilde{\alpha}_g$, $\bigwedge_{j \in MSG(seq_1)} \rho_j$, and $\bigwedge_{j \in MSG(seq_1)} \beta_j$.

The proof follows the the one in case 2.1.

Then, we consider the semantic aspects of the OSs within each Operand of cf_1. The semantic aspects specify that only the order of the OSs within each Operand whose Constraint evaluates to *True* is maintained. The Operands whose Constraints evaluate to *False* are excluded. Each Operand can be considered as a basic Sequence Diagram with Constraint. The OSs within each Operand respect the same order as the OSs within a basic Sequence Diagram. Sub-formula $\tilde{\theta}^{cf_1}$ describes the semantics of the Operands whose Constraints evaluate to *True* using function $TOP(cf_1)$, where the formula for each Operand follows the formula for a basic Sequence Diagram, *i.e.*, a conjunction of $\tilde{\alpha}_a$s, β_js, and ρ_js. Therefore, we can prove that $\upsilon \cdot \tau^{\omega} \models \tilde{\theta}^{cf_1}$ by following the proof of basic Sequence Diagram.

Next, we consider the semantic aspects which describe the order between cf_1 and its adjacent OSs. cf_1 and its adjacent OSs are connected using Weak Sequencing, *i.e.*, for Lifeline $i(1 \leq i \leq j)$, cf_1's preceding OSs must execute before its CEU's execution, and cf_1's succeeding OSs must execute afterwards. Function $pre(cf_1 \uparrow_i)$ returns the set of OSs which may happen right before CEU $cf_1 \uparrow_i$. The semantics aspect of seq_1 defines that, for Lifeline $i(1 \leq i \leq j)$, any OS within $cf_1 \uparrow_i$ cannot execute until all OSs in $pre(cf_1 \uparrow_i)$ finish execution. We wish to prove that the semantic aspect is captured by the first conjunct of subformula $\tilde{\gamma}^{cf_1}$. We assume that, if $\upsilon \cdot \tau^{\omega}$ does not satisfy the first conjunct of $\tilde{\gamma}^{cf_1}$, then $\tilde{\gamma}^{cf_1}$ defines that, on Lifeline i, at least one OS, r_{c+d} (see figure B.5), occurs before OS_{pre}. OS_{pre} is an OS in $pre(cf_1 \uparrow_i)$. The first conjunct of $\tilde{\gamma}^{cf_1}$ specifies that any OS within cf_1 on Lifeline i cannot execute until the OSs in $pre(cf_1 \uparrow_i)$ finish execution, so OS_{pre} must happen before r_{c+d}, which contradicts our assumption. In this way, we can prove that $\upsilon \cdot \tau^{\omega}$ satisfies the first conjunct of $\tilde{\gamma}^{cf_1}$. Similarly, we can also prove that $\upsilon \cdot \tau^{\omega}$ satisfies the second conjunct of $\tilde{\gamma}^{cf_1}$. Hence, $\upsilon \cdot \tau^{\omega} \models \tilde{\gamma}^{cf_1}$.

Finally, we consider the semantic aspect for the seq_1. We define the OSs which are directly enclosed in seq_1 or Operands whose Constraints evaluate to *True* as enabled OS, *i.e.*, these OSs can be enabled to occur. Function $AOS(seq_1)$ returns the set of enabled OSs within seq. The semantic aspect specifies that only one enabled OS can execute at a time, and all the enabled OSs should execute uninterrupted. If $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, we can deduce that $|\upsilon| = |AOS(seq_1)| = 2h + 2p$. It is easy to infer that $\upsilon \cdot \tau^{\omega} \models \varepsilon_{seq_1}$.

Now we have proven that if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_1}$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^{*}$.

If $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, then $\sigma = \sigma_{[1..2h+2p]} \cdot \tau^{\omega}$, which follows Lemma 4.10. We wish to prove that $\sigma_{[1..2h+2p]}$ respects all the semantics of seq_1. $\sigma \models \widetilde{\Pi}_{seq_1}$, so σ satisfies all sub-formulas of $\widetilde{\Pi}_{seq_1}$. We prove that the sub-formulas capture the semantic aspects as below.

First, we discuss the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j for seq_1. We can prove that these sub-formulas capture the semantics of OSs directly enclosed in seq_1. The proof follows the one in case 2.1.

Then, we discuss the sub-formula $\tilde{\theta}^{cf_1}$. Function $\bigwedge\limits_{op \in TOP(cf_1)}$ returns the set of Operands whose Constraints evaluate to *True* within cf_1. Hence, $\tilde{\theta}^{cf_1}$ only captures the semantics of Operands whose Constraints evaluate to *True*. It is consistent with the semantic aspect of cf_1, which excludes the Operands whose Constraints evaluate to *False*. For each Operand whose Constraints evaluate to *True*, we wish to prove that sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j capture its semantics. cf_1 contains no other CFs, so $ABEU(op \uparrow_i)$ returns the BEU of op on Lifeline i. We can consider an Operand with no nested CFs as a basic Sequence Diagram with Interaction Constraint. In this way, we can prove that these sub-formulas capture the Operand's semantics by following the proof of basic Sequence Diagram. Therefore, we have proven that $\tilde{\theta}^{cf_1}$ captures the semantics of Combined Fragment cf_1.

Next, we discuss the sub-formula $\tilde{\gamma}_i^{cf_1}$ for Lifeline i. We wish to prove that it captures the order of CEU $cf_1 \uparrow_i$ and its preceding/succeeding OSs on Lifeline i. The first conjunct of $\tilde{\gamma}_i^{cf_1}$ defines that any OS in CEU $cf_1 \uparrow_i$ cannot happen before all OSs in $pre(cf_1 \uparrow_i)$ finish execution. If it does not capture the semantic aspect, then we assume that at least an OS in $pre(cf_1 \uparrow_i)$, OS_{pre}, occurs after an OS in $cf_1 \uparrow_i$, r_{c+d}. Function $pre(cf_1 \uparrow_i)$ returns the set of OSs which may happen right before CEU $cf_1 \uparrow_i$. The semantics defines that all OS in $pre(cf_1 \uparrow_i)$ must happen before all OS within CEU $cf_1 \uparrow_i$. Thus, OS_{pre} must occur before

r_{c+d}, which contradicts our assumption. In this way, we have proven that the first conjunct of $\tilde{\gamma}_i^{cf_1}$ captures the order of CEU $cf_1 \uparrow_i$ and its preceding OSs on Lifeline i. Similarly, we can prove that the second conjunct of $\tilde{\gamma}_i^{cf_1}$ captures the order of CEU $cf_1 \uparrow_i$ and its succeeding OSs on Lifeline i. Therefore, we have proven that $\tilde{\gamma}_i^{cf_1}$ captures the order of CEU $cf_1 \uparrow_i$ and its preceding/succeeding OSs on Lifeline i.

Finally, we discuss the sub-formula ε_{seq_1}. It represents that only one OS in $|AOS(seq_1)|$ executes at a time, or all OSs in $|AOS(seq_1)|$ have executed. Function $|AOS(seq_1)|$ returns the set of OSs which can be enabled to execute in seq_1, *i.e.*, it returns a set which includes the OSs directly enclosed in seq_1 and the OSs within cf_1's Operand whose Constraint evaluates to *True*. In seq_1, $|AOS(seq_1)| = 2h + 2p$. From lemma 4.10, if $\sigma \models \varepsilon_{seq_1}$, then $\sigma = \sigma_{[1..2h+2p]} \cdot \tau^{\omega}$. Therefore, ε_{seq_1} captures the semantic aspect, which enforces that only one object can execute an OS at a time and all enabled OSs of seq_1 execute uninterrupted.

Now we have proven that $\forall \sigma . \sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, respects all the semantic aspects of seq_1, *i.e.*, $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

If cf_1 contains more than two Operands, p Messages may be enclosed in multiple Operands whose Interaction Constraints evaluate to *True*, and q Messages may be enclosed in multiple Operands whose Interaction Constraints evaluate to *False*. The proof follows the one for cf_1 with two Operands.

To conclude, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_1}$, and $\forall \sigma . \sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^{\omega}$, then $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

We have proven that the semantic rules general to all CFs can be captured by our LTL templates. The semantic rules for each CF with different Operators can be enforced by adding different semantic constraints, which are captured using LTL template ϱ^{CF}. Parallel defines that the OSs within different Operands may be interleaved. Its semantics does not introduce additional semantic rule. Thus, we have proven that our LTL templates capture the semantics of Parallel.

We use Strict Sequencing as an example to prove that the semantic rule for each Operator can be captured by our LTL templates. The cases for CFs with other Operators can be proven similarly.

$*$ Case 2.2.2 We assume that, a given Strict Sequencing, cf_1^{strict}, has two Operands whose Interaction Constraints evaluate to ***True***. The first Operand contains p_1 Messages, and the second Operand contains p_2 Messages. cf_1^{strict} covers i Lifelines.

(a) We wish to prove that, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

The Strict Sequencing imposes an order among OSs within different Operands. For an Operand (not the first Operand), any OS cannot occur before the OSs within the previous Operand finish execution. Function $preEU(u)$ returns the set of OSs within EU v which happen right before EU u, *i.e.*, the Constraint of EU v evaluates to ***True***. In this case, $preEU(u)$ returns the last OS in EU u. The semantic aspect of Strict Sequencing can be considered as that, any OS in Operand k cannot happen until the OSs in all $preEU(u)$, where u is the EU of Operand k on Lifeline $j (1 \leq j \leq i)$, finish execution. We introduce sub-formula χ_k to capture the semantics of Operand k. We assume that, if $\upsilon \cdot \tau^\omega$ does not satisfies $\bigwedge_{k \in NFTOP(cf_1^{strict})} \chi_k$, then χ_k defines that at least one OS, OS_s, in Operand k, occurs before OS_{pre}, which is an OS in $preEU((k-1) \uparrow_j)$, where $1 \leq j \leq i$. Sub-formula χ_k specifies that any OS within $preEU(u)$ on all the Lifelines covered by the Strict Sequencing must happen before the OSs within Operand k. Therefore, OS_{pre} must happen before OS_s, which contradicts our assumptions. Thus, we can prove that $\upsilon \cdot \tau^\omega \models \bigwedge_{k \in NFTOP(cf_1^{strict})} \chi_k$.

We have proven that $\upsilon \cdot \tau^\omega$ satisfies other general sub-formulas of Π_{seq_1} in case 2.1.2(1). Hence, we can prove that $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, $\sigma_{[1..2h+2p_1+2p_2]} \in (\Sigma_{sem}^{seq_1})^*$.

If $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, then $\sigma = \sigma_{[1..2h+2p_1+2p_2]} \cdot \tau^\omega$, which follows Lemma 4.10. We wish to prove that $\sigma_{[1..2h+2p_1+2p_2]}$ respects all the semantics of seq_1. $\sigma \models \widetilde{\Pi}_{seq_1}$, so σ satisfies all sub-formulas of $\widetilde{\Pi}_{seq_1}$. We have proven that the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j capture the semantics of OS directly enclosed in seq_1; sub-formulas $\tilde{\theta}^{cf_1^{strict}}$ and $\tilde{\gamma}_i^{cf_1^{strict}}$ capture the general semantic aspects of cf_1^{strict}; sub-formula ε_{seq_1} captures the interleaving semantics of seq_1 (see case 2.1.2(1)). Now we need to prove that sub-formula $\bigwedge\limits_{k \in NFTOP(cf_1^{strict})} \chi_k$ captures the semantics of Strict Sequencing.

Sub-formula $\bigwedge\limits_{k \in NFTOP(cf_1^{strict})} \chi_k$ asserts the order between each Operand k of Strict Sequencing (k is not the first Operand), and its preceding Operand. Function $preEU(u)$ returns the set of OSs within EU v which happen right before EU u. Each OS within k cannot happen until all OS within $preEU(u)$ on all the Lifelines covered by the Strict Sequencing. If the sub-formula does not capture the semantics of Strict Sequencing, we assume the semantics defines that at least an OS in $preEU((k-1)\uparrow_j)(1 \leq j \leq i)$, OS_{pre}, occurs after an OS in Operand k, OS_s. Actually, the semantics of Strict Sequencing defines that in any Operand except the first one, OSs cannot execute until the previous Operand completes. Therefore, OS_{pre} must happen before OS_s, which contradicts our assumption. In this way, we can prove that sub-formula $\bigwedge\limits_{k \in NFTOP(cf_1^{strict})} \chi_k$ captures the semantics of Strict Sequencing. Hence, we can prove that $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

To conclude, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_1}$, and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, then $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

– Case 2.3 The semantics of Alternatives defines that at most one of its Operand whose Constraints evaluate to *True* is chosen to execute. The Operands whose Constraints evaluate to *False* are still excluded. To capture its semantics, we need to specify the semantics of the chosen Operand and the connection between the chosen Operand and its adjacent OSs. We use LTL formula Ψ_{alt}^{CF} to capture the semantic of Alternatives. Sub-formulas $\bar{\theta}_m^{CF}$ and $\bar{\gamma}_{i,m}^{CF}$ can be rewritten into $\bar{\tilde{\theta}}_m^{CF}$ and $\bar{\tilde{\gamma}}_{i,m}^{CF}$ by following the same procedures of rewriting sub-formulas θ^{CF} and γ_i^{CF}. The LTL formula of Alternatives, Ψ_{alt}^{CF}, with rewritten sub-formulas is shown in figure B.6.

$$
\begin{aligned}
\Psi_{alt}^{CF} &= \bigwedge_{m \in TOP(CF)} \Psi_{alt}^{m} \\
\Psi_{alt}^{m} &= \begin{cases} \bar{\tilde{\theta}}_m^{CF} \wedge \bigwedge\limits_{i \in LN(CF)} \bar{\tilde{\gamma}}_{i,m}^{CF} \wedge \bigwedge\limits_{CF_t \in nested(m)} \Phi^{CF_t} & if\ m\ is\ the\ chosen\ Operand \quad (1) \\ True & else \quad (2) \end{cases} \\
\bar{\tilde{\theta}}_m^{CF} &= \Big(\bigwedge_{\substack{i \in LN(m) \\ g \in ABEU(m\uparrow_i)}} \tilde{\alpha}_g \Big) \wedge \Big(\bigwedge_{j \in MSG(m)} \rho_j \Big) \wedge \Big(\bigwedge_{j \in MSG(m)} \beta_j \Big) \Big) \\
\bar{\tilde{\gamma}}_{i,m}^{CF} &= \bigwedge_{\substack{beu \in ABEU(m\uparrow_i) \\ OS \in AOS(beu)}} \Big((\neg OS\, \widetilde{\mathcal{U}}\, \Big(\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\diamondsuit OS_{pre}) \Big) \Big) \wedge \Big(\Big(\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post}) \Big) \widetilde{\mathcal{U}}\, (\diamondsuit OS) \Big) \Big)
\end{aligned}
$$

Figure B.6: Rewriting LTL formula for Alternatives

We assume that, a given Alternatives, cf_1^{alt}, has two Operands whose Interaction Constraints evaluate to *True*. The first Operand contains p_1 Messages, and the second Operand contains p_2 Messages. cf_1^{alt} covers i Lifelines.

$$\Phi^{cf_1} = \Psi_{alt}^{cf_1}$$

(a) We wish to prove that, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

For Alternatives. We only consider the Operands whose Constraints evaluate to *True* as defined by the general semantics rules. If more than one Operand's Constraint evaluates to *True*, at most one Operand is chosen and the order of the OSs within it should be specified. Sub-formula Ψ_{alt}^m defines the semantics of Operand m whose Constraint evaluates to *True*. If m is chosen, its semantics is captured by sub-formula

$\bar{\tilde{\theta}}_m^{cf_1^{alt}}$ and $\bar{\tilde{\gamma}}_{i,m}^{cf_1^{alt}}$. Otherwise, Ψ_{alt}^m evaluates to *True*, denoting that m is excluded. We can prove that $\bar{\tilde{\theta}}_m^{cf_1^{alt}}$ describes the order among OSs within m by following the proof for sub-formula $\tilde{\theta}^{cf_1^{alt}}$. Similarly, we can prove that $\bar{\tilde{\gamma}}_{i,m}^{cf_1^{alt}}$ describes the order among OSs within m and the Alternatives's adjacent OSs by following the proof for sub-formula $\tilde{\gamma}_i^{cf_1^{alt}}$. Therefore, $\upsilon \cdot \tau^\omega$ satisfies $\Psi_{alt}^{cf_1^{alt}}$.

We have proven that $\upsilon \cdot \tau^\omega$ satisfies $\tilde{\alpha}_g$, ρ_j, and β_j for seq_1 in case 2.1.2(1). For sub-formula ε_{seq_1}, function $AOS(seq_1)$ returns the enabled and chosen OSs, *i.e.*, for Alternatives, only the OSs within the chosen Operand are returned. We can prove that $\upsilon \cdot \tau^\omega$ satisfies ε_{seq_1} by following the proof in case 2.1.2(1). Hence, we can prove that $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

(b) We wish to prove that, $\forall \sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, $\sigma_{[1..2h+2p_m]} \in (\Sigma_{sem}^{seq_1})^*$ (m is the chosen Operand of cf_1^{alt}).

If $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, then $\sigma = \sigma_{[1..2h+2p_m]} \cdot \tau^\omega$, which follows Lemma 4.10. We wish to prove that $\sigma = \sigma_{[1..2h+2p_m]}$ respects all the semantics of seq_1. $\sigma \models \widetilde{\Pi}_{seq_1}$, so σ satisfies all sub-formulas of $\widetilde{\Pi}_{seq_1}$. We have proven that the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j capture the semantics of OS directly enclosed in seq_1; sub-formula ε_{seq_1} captures the interleaving semantics of seq_1 (see case 2.1.2(1)). We need to prove that sub-formula $\Psi_{alt}^{cf_1^{alt}}$ captures the semantics of Alternatives.

Sub-formula $\Psi_{alt}^{cf_1^{alt}}$ is a conjunction of sub-formula Ψ_{alt}^ms, where m is an Alternatives's Operand whose Constraint evaluates to *True*. Therefore, the Operands whose Constraints evaluate to *False* are excluded. Ψ_{alt}^m evaluates to *False* if m is unchosen, which captures the semantics that the unchosen Operands are excluded. Ψ_{alt}^m is a conjunction of sub-formulas $\bar{\tilde{\theta}}_m^{cf_1^{alt}}$ and $\bar{\tilde{\gamma}}_{i,m}^{cf_1^{alt}}$ when m is the chosen Operand. We can prove that sub-formula $\bar{\tilde{\theta}}_m^{cf_1^{alt}}$ captures the order among OSs within m by following the proof of $\tilde{\theta}_m^{cf_1^{alt}}$. We can also prove that sub-formula $\bar{\tilde{\gamma}}_{i,m}^{cf_1^{alt}}$ captures the order between OSs within m and the Alternatives's adjacent OSs by following the proof of $\tilde{\gamma}_i^{cf_1^{alt}}$. In this way, we can prove that sub-formula $\Psi_{alt}^{cf_1^{alt}}$ captures the semantics of Alternatives. Hence, we can prove that $\sigma_{[1..2h+2p_m]} \in (\Sigma_{sem}^{seq_1})^*$.

To conclude, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_1}$, and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_1})^\omega$, then $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

- Case 2.4 The Loop represents the iterations of its sole Operand. We capture the semantics of Loop using LTL formula $\Psi_{loop,R}^{CF}$, which unfolds the iterations and connects them using Weak Sequencing. Each iteration can be considered as an Operand, whose semantics can be captured by sub-formulas α_g, β_j, ρ_j and γ_i as proven. We need to prove that sub-formula $\bigwedge\limits_{i \in LN(CF)} \kappa_{i,R}$ captures the Weak Sequencing among iterations. The proof is quite similar as the proof for sub-formula $\bigwedge\limits_{k \in NFTOP(CF)} \chi_k$ of Strict Sequencing.

Inductive step. A given Sequence Diagram, seq_n, directly contains n CFs. For the Messages within the CFs, p_n Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We assume $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_n})^*$, then $\upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_n}$. $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{LTL}^{seq_n})^\omega$, then $\sigma_{[1..2h+2p_n]} \in (\Sigma_{sem}^{seq})^*$. $(r = n)$

We add a CF, cf_{n+1}, in seq_n to form a new Sequence Diagram, seq_{n+1}, with $n{+}1$ CFs. cf_{n+1} is directly enclosed in seq_{n+1}. In cf_{n+1}, p_{n+1} Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We wish to prove that, $\forall \upsilon' . \upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{n+1}}$. $\forall \sigma' . \sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

The LTL templates $\widetilde{\Pi}_{seq_n}$ and $\widetilde{\Pi}_{seq_{n+1}}$ are shown as,

$$\widetilde{\Pi}_{seq_n} = (\bigwedge_{\substack{i \in LN(seq_n)\\ g \in TBEU(seq_n \uparrow_i)}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq_n)} \rho_j) \wedge (\bigwedge_{j \in MSG(seq_n)} \beta_j) \wedge (\bigwedge_{CF \in nested(seq_n)} \Phi^{CF}) \wedge \varepsilon_{seq_n}$$

$$\widetilde{\Pi}_{seq_{n+1}} = (\bigwedge_{\substack{i \in LN(seq_{n+1})\\ g \in TBEU(seq_{n+1} \uparrow_i)}} \tilde{\alpha}_g) \wedge (\bigwedge_{j \in MSG(seq_{n+1})} \rho_j) \wedge (\bigwedge_{j \in MSG(seq_{n+1})} \beta_j) \wedge (\bigwedge_{CF \in nested(seq_n)} \Phi^{CF}) \wedge \Phi^{cf_{n+1}} \wedge \varepsilon_{seq_{n+1}}$$

(a) We wish to prove that, $\forall \upsilon' . \upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{n+1}}$.

First, we consider the semantic aspects of the OSs directly enclosed in seq_{n+1}. We can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies $\bigwedge_{\substack{i \in LN(seq_{n+1}) \\ g \in TBEU(seq_{n+1} \uparrow_i)}} \tilde{\alpha}_g$, $\bigwedge_{j \in MSG(seq_{n+1})} \rho_j$, and $\bigwedge_{j \in MSG(seq_{n+1})} \beta_j$. The proof follows the the one in case 2.1 of basic case.

Then, we consider the semantic aspects of n CFs, which are captured by LTL formula $\bigwedge_{CF \in nested(seq_n)} \Phi^{CF}$ in seq_n. The newly added CF is directly enclosed in seq_{n+1}, so it does not interact with the existing CFs. Therefore, in seq_{n+1}, the semantics of existing CFs can still be captured by formula $\bigwedge_{CF \in nested(seq_n)} \Phi^{CF}$. We can prove that $\upsilon' \cdot \tau^{\omega} \models \bigwedge_{CF \in nested(seq_n)} \Phi^{CF}$.

Next, we consider the semantic aspects of cf_{n+1}, which is captured using formula $\Phi^{cf_{n+1}}$. For $\Phi^{cf_{n+1}}$, sub-formulas $\tilde{\theta}^{cf_{n+1}}$, $\tilde{\gamma}_i^{cf_{n+1}}$, and the additional ones for each Operator still define the semantics we have proven in base case. The order of OSs within each CF is not changed. Therefore, $\upsilon' \cdot \tau^{\omega}$ satisfies $\tilde{\theta}^{cf_{n+1}}$ and the additional sub-formulas for each Operand. Sub-formula $\tilde{\gamma}_i^{cf_{n+1}}$ still specifies the Weak Sequencing between cf_{n+1} and its preceding/succeeding Interaction Fragments. Comparing to base case, cf_{n+1}'s preceding/succeeding Interaction Fragments can be OSs or CFs. We wish to prove that our algorithms for calculating $pre(cf_{n+1} \uparrow_i)$ and $post(cf_{n+1} \uparrow_i)$ are correct.

Function $pre(cf_{n+1} \uparrow_i)$ returns the set of OSs which happen right before CEU $cf_{n+1} \uparrow_i$. We focus on the CEU or EU υ prior to $cf_{n+1} \uparrow_i$ on Lifeline i. The EUs whose Constraints evaluate to *False* are excluded. Therefore, υ should be a CEU containing at least one EU whose Constraint evaluates to *True* or an EU whose Constraint evaluates to *True*. We start from the CEU or EU prior to $cf_{n+1} \uparrow_i$, and check the CEUs and EUs until we find υ. If υ does not exist, we define that the first conjunct of $\tilde{\gamma}_i^{cf_{n+1}}$ evaluates to *True*. Otherwise, we discuss the return value of the function by different cases.

- Case i. If υ is a BEU, function returns the OS in the bottom of υ, OS_t. We assume that if the function returns another OS, OS_s, then OS_s should happen after OS_t. However, the semantics defines that OSs are ordered graphically in a BEU. OS_t is the last one to execute in υ, which contradicts our assumption. Thus, we can prove that the function returns the OS in the bottom of v.

- Case ii. If v is a CEU with one BEU whose Constraint evaluates to *True*, function returns the OS in the bottom of the BEU as we proven in case 1.

- Case iii. If v is a CEU with multiple BEUs whose Constraints evaluate to *True*. (1) v with Operator "par" returns a set containing the last OS of each BEU, as defined by the semantics of Parallel (We have proven in base case 2.2.1); (2) v with Operator "alt" returns a set containing the last OS of the chosen BEU, as defined by the semantics of Alternatives (We have proven in base case 2.3); (3) v with Operator "weak" or "strict" returns a set containing the last OS of the last BEU, as defined by the semantics of Strict Sequencing (We have proven in base case 2.2.2).

- Case iv. If v is a CEU with nested CEUs. (1) If v directly contains only one EU whose Constraint evaluates to *True*, we find the EU's last CEU or EU, w, and recursively apply case 1 to 4 to prove it. (2) If v directly contains multiple EUs whose Constraint evaluates to *True*, we recursively apply case 1 to 4 to (a) each EU to prove it (v's Operator is "par"); (b) the chosen EU to prove it (v's Operator is "alt"); (c) the last EU (v's Operator is "weak" or "strict") to prove it.

The proof of the algorithm for calculating $post(cf_{n+1} \uparrow_i)$ follows the one of the algorithm for calculating $pre(cf_{n+1} \uparrow_i)$. Hence, $\upsilon \cdot \tau^\omega \models \tilde{\gamma}^{cf_{n+1}}$.

Finally, we consider the semantic aspect for the seq_{n+1}. Function $AOS(seq_{n+1})$ returns the set of chosen and enabled OSs within seq_{n+1}. The semantic aspect specifies that only one enabled OS can execute at a time, and all enabled OSs should execute uninterrupted. If $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, we can deduce that $|\upsilon'| = |AOS(seq_{n+1})| = 2h+2p_n+2p_{n+1}$. It is easy to infer that $\upsilon' \cdot \tau^\omega \models \varepsilon_{seq_{n+1}}$.

Now we have proven that if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{n+1}}$.

(b) We wish to prove that, $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

If $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma' = \sigma_{[1..2h+2p_n+2p_{n+1}]} \cdot \tau^\omega$, which follows Lemma 4.10. We wish to prove that $\sigma'_{[1..2h+2p_n+2p_{n+1}]}$ respects all the semantics of seq_{n+1}. $\sigma' \models \widetilde{\Pi}_{seq_{n+1}}$, so σ' satisfies all sub-formulas of $\widetilde{\Pi}_{seq_{n+1}}$. We prove that the sub-formulas capture the semantic aspects as below.

First, we discuss the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j for seq_{n+1}. We can prove that these sub-formulas capture the semantics of OSs directly enclosed in seq_{n+1}. The proof follows the one in case 2.1.

Then, we discuss the sub-formula $\bigwedge\limits_{CF \in nested(seq)} \Phi^{CF}$. In seq_n, the sub-formula captures the semantics of n CFs. In seq_{n+1}, adding cf_{n+1} does not change the semantics of the existing CFs. It is easy to infer that, sub-formula $\bigwedge\limits_{CF \in nested(seq)} \Phi^{CF}$ still captures the semantics of the CFs except for cf_{n+1}.

Next, we discuss the sub-formula formula $\Phi^{cf_{n+1}}$, which is a conjunction of sub-formulas $\tilde{\theta}^{cf_{n+1}}$, $\tilde{\gamma}_i^{cf_{n+1}}$, and the additional one for each Operator. With the proof of base case, $\tilde{\theta}^{cf_{n+1}}$ captures the semantics of cf_{n+1}'s Operands, while the additional sub-formula captures the semantics of cf_{n+1}'s Operator. Sub-formula $\tilde{\gamma}_i^{cf_{n+1}}$ may be different from the base case, since the preceding/succeeding Interaction Fragment of cf_{n+1} can be other CFs. On Lifeline i, functions $pre(cf_{n+1} \uparrow_i)$ and $post(cf_{n+1} \uparrow_i)$ return the set of OSs which may happen right before and after CEU $cf_{n+1} \uparrow_i$ respectively. We have proven that our algorithms can calculate $pre(cf_{n+1} \uparrow_i)$ and $post(cf_{n+1} \uparrow_i)$ for all the cases. Thus, we can infer that $\bigwedge\limits_{i \in LN(CF)} \tilde{\gamma}_i^{cf_{n+1}}$ still captures the Weak Sequencing between cf_{n+1} and its preceding/succeeding Interaction Fragments.

Finally, we discuss the sub-formula $\varepsilon_{seq_{n+1}}$. It represent only one OS in $|AOS(seq_{n+1})|$ execute at a time, or all OSs in $|AOS(seq_{n+1})|$ have executed. Function $|AOS(seq_{n+1})|$ returns the set of OSs which are chosen and enabled to execute in seq_{n+1}. In seq_{n+1}, $|AOS(seq_{n+1})| = 2h + 2p_n + 2p_{n+1}$. From lemma 4.10, if $\sigma \models \varepsilon_{seq_{n+1}}$, then $\sigma = \sigma_{[1..2h+2p_n+2p_{n+1}]} \cdot \tau^{\omega}$. Therefore, $\varepsilon_{seq_{n+1}}$ captures the interleaving semantics of seq_{n+1}.

Now we have proven that $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^{\omega}$, respects all the semantic aspects of seq_{n+1}, *i.e.*, $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

To conclude, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_{n+1}}$, and $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^{\omega}$, then $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$. $\square$

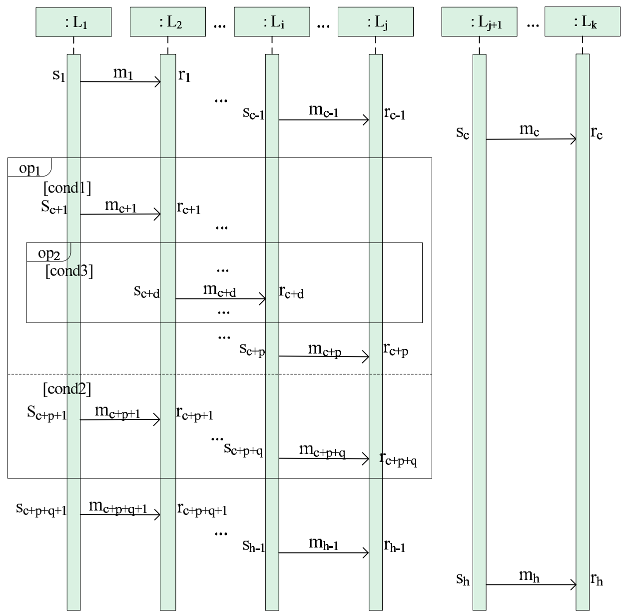


Figure B.7: Example of Sequence Diagram with nested CF

If a Sequence Diagram contains nested CFs, the semantics of nested CFs are independent. For instance, if cf_1 whose Operand is op_1 contains cf_2 whose Operand is op_2 (see figure B.7), the semantic rules of cf_1 do not constraint the semantic rules of cf_2.

Theorem 4.12. $(\Sigma_{sem}^{seq_{nested}})^*$ and $PRE_{2h+2p}((\Sigma_{LTL}^{seq_{nested}})^\omega)$ are equal.

Proof. We use mathematical induction, which is based on the maximal layer of CF, l, within seq_{nested}.

Base step. seq_{nested} directly contains r CFs, each of which does not contain other CFs. $(l = 1)$

The proof follows the one for theorem 4.11.

Inductive step. seq_n^{nested} directly contains r CFs. We assume that cf_v, which is a CF directly enclosed in seq_n^{nested}, contains cf_w, which is a CF with the maximal layer within seq_n^{nested}. The maximal layer of CF within seq_n^{nested} is n. For the Messages within the CFs, p_n Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We assume $\forall \upsilon . \upsilon \in (\Sigma_{sem}^{seq_n^{nested}})^*, \upsilon \cdot \tau^\omega \models \widetilde{\Pi}_{seq_n^{nested}}$. $\forall \sigma . \sigma \in (\Sigma_{LTL}^{seq_n^{nested}})^\omega$, then $\sigma_{[1..2h+2p_n]} \in (\Sigma_{sem}^{seq_n^{nested}})^*$. $(l = n)$

We add a CF, cf_u, in seq_n^{nested} to form a new Sequence Diagram, seq_{n+1}^{nested}, where cf_u contains

cf_v. The layer of cf_w becomes $n+1$, which is the maximal layer of CF within seq_{n+1}^{nested}. In seq_{n+1}^{nested}, p_{n+1} Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We wish to prove that, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$, then $\upsilon' \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{n+1}^{nested}}$. $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}^{nested}})^\omega$, then $\sigma'_{[1..2n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$.

When we add cf_u into seq_n^{nested}, then order of the OSs directly enclosed in seq_n^{nested} keep unchanged. Thus, the semantics of the OSs directly enclosed in seq_{n+1}^{nested} can still be captured using the corresponding sub-formulas of $\widetilde{\Pi}_{seq_n^{nested}}$. The LTL templates $\widetilde{\Pi}_{seq_n^{nested}}$ and $\widetilde{\Pi}_{seq_{n+1}^{nested}}$ are shown as,

$$\begin{aligned}
\widetilde{\Pi}_{seq_n^{nested}} =& \Big(\bigwedge_{\substack{i\in LN(seq_n^{nested})\\ g\in TBEU(seq_n^{nested}\uparrow_i)}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_n^{nested})} \rho_j\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_n^{nested})} \beta_j\Big) \wedge \Big(\bigwedge_{CF\in nested(seq_n^{nested})} \Phi^{CF}\Big) \\
&\wedge \varepsilon_{seq_n^{nested}} \\
\widetilde{\Pi}_{seq_{n+1}^{nested}} =& \Big(\bigwedge_{\substack{i\in LN(seq_{n+1}^{nested})\\ g\in TBEU(seq_{n+1}^{nested}\uparrow_i)}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_{n+1}^{nested})} \rho_j\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_{n+1}^{nested})} \beta_j\Big) \wedge \Big(\bigwedge_{CF\in nested(seq_{n+1}^{nested})} \Phi^{CF}\Big) \\
&\wedge \varepsilon_{seq_{n+1}^{nested}} \\
=& \Big(\bigwedge_{\substack{i\in LN(seq_n^{nested})\\ g\in TBEU(seq_n^{nested}\uparrow_i)}} \tilde{\alpha}_g\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_n^{nested})} \rho_j\Big) \wedge \Big(\bigwedge_{j\in MSG(seq_n^{nested})} \beta_j\Big) \wedge \Big(\bigwedge_{\substack{CF\in nested(seq_n^{nested})\\ CF\neq cf_v}} \Phi^{CF}\Big) \\
&\wedge \Phi^{cf_u} \wedge \varepsilon_{seq_{n+1}^{nested}}
\end{aligned}$$

(a) We wish to prove that, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$, then $\upsilon' \cdot \tau^\omega \models \widetilde{\Pi}_{seq_{n+1}^{nested}}$.

We wish to prove that $\upsilon' \cdot \tau^\omega$ satisfies all sub-formulas of $\widetilde{\Pi}_{seq_{n+1}^{nested}}$.

First, we consider the OSs directly enclosed in seq_{n+1}^{nested}. The semantics of the OSs directly enclosed in seq_n^{nested} are not altered by adding cf_u. Thus, we can prove that $\upsilon' \cdot \tau^\omega$ satisfies the sub-formulas of $\widetilde{\Pi}_{seq_{n+1}^{nested}}$ capturing the semantics of the OSs directly enclosed in seq_{n+1}^{nested}, *i.e.*, $\bigwedge_{\substack{i\in LN(seq_n^{nested})\\ g\in TBEU(seq_n^{nested}\uparrow_i)}} \tilde{\alpha}_g$, $\bigwedge_{j\in MSG(seq_n^{nested})} \rho_j$, and $\bigwedge_{j\in MSG(seq_n^{nested})} \beta_j$.

Then, we consider the CFs (except cf_u) directly enclosed in seq_{n+1}^{nested}. The semantics of these CFs and the LTL sub-formulas capturing their semantics are not changed. It is easy to infer that

$\upsilon' \cdot \tau^{\omega}$ satisfies $\bigwedge_{\substack{CF \in nested(seq_n^{nested}) \\ CF \neq cf_v}} \Phi^{CF}$.

Next, we consider CF cf_u, whose semantics is captured using Φ^{cf_u}. We discuss sub-formula Φ^{cf_u} using three cases.(1) If all the Constraints of cf_u's Operands evaluate to *False*, $\Phi^{cf_u} = \eta^{cf_u}$. We can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies Φ^{cf_u}. The proof follows the one for base case. (2) If not all the Constraints of cf_u's Operands evaluate to *False*, and the Operator of cf_u is not alt or $loop$, $\Phi^{cf_u} = \Psi^{cf_u} \wedge \Phi^{cf_v}$. The semantics of cf_v is not altered by adding cf_u. Hence, we can infer that $\upsilon' \cdot \tau^{\omega}$ satisfies Φ^{cf_v}. $\Psi^{cf_u} = \tilde{\theta}^{cf_u} \wedge \bigwedge_{i \in LN(cf_u)} \tilde{\gamma}_i^{cf_u}$. We can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies $\tilde{\theta}^{cf_u}$ and $\bigwedge_{i \in LN(cf_u)} \tilde{\gamma}_i^{cf_u}$. The proof follows the one for base case. (3) If not all the Constraints of cf_u's Operands evaluate to *False*, and the Operator of cf_u is alt or $loop$, $\Phi^{cf_u} = \Psi_{alt}^{cf_u}$ or $\Phi^{cf_u} = \Psi_{loop}^{cf_u}$ respectively. Similarly, we can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies $\Psi_{alt}^{cf_u}$ or $\Psi_{loop}^{cf_u}$.

Finally, we consider the interleaving semantics of seq_{n+1}^{nested}. Function $AOS(seq_{n+1}^{nested})$ returns the set of chosen and enabled OSs within seq_{n+1}^{nested}. Sub-formula $\varepsilon_{seq_{n+1}^{nested}}$ specifies that only one OS execute at a state, or all OS have executed. If $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$, we can deduce that $|\upsilon'| = |AOS(seq_{n+1}^{nested})| = 2h + 2p_{n+1}$. It is easy to infer that $\upsilon' \cdot \tau^{\omega} \models \varepsilon_{seq_{n+1}^{nested}}$.

Now we have proven that if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$, then $\upsilon' \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_{n+1}^{nested}}$.

(b) We wish to prove that, $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}^{nested}})^{\omega}$, then $\sigma'_{[1..2n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$.

If $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}^{nested}})^{\omega}$, then $\sigma' = \sigma_{[1..2h+2p_{n+1}]} \cdot \tau^{\omega}$, which follows Lemma 4.10. We wish to prove that $\sigma'_{[1..2h+2p_{n+1}]}$ respects all the semantics of $seq_{seq_{n+1}^{nested}}$. $\sigma' \models \widetilde{\Pi}_{seq_{n+1}^{nested}}$, so σ' satisfies all sub-formulas of $\widetilde{\Pi}_{seq_{n+1}^{nested}}$. We prove that the sub-formulas capture the semantic aspects as below.

First, we discuss the sub-formulas $\tilde{\alpha}_g$, ρ_j, and β_j for seq_{n+1}^{nested}. These sub-formulas are not changed, so they still capture the semantics of OSs directly enclosed in seq_n^{nested}. We can also infer that these sub-formulas capture the semantics of OSs directly enclosed in seq_{n+1}^{nested}

Then, we discuss the sub-formula $\bigwedge_{\substack{CF \in nested(seq_n^{nested}) \\ CF \neq cf_v}} \Phi^{CF}$. For seq_n, the sub-formula captures the semantics of the CFs (except for cf_v) directly enclosed in it. In seq_{n+1}, adding cf_u does not change the semantics of the CFs except for cf_v. It is easy to infer that, the sub-formula still captures the semantics of the CFs except for cf_v.

Next, we discuss the sub-formula formula Φ^{cf_u} using three cases. (1)$\Phi^{cf_u} = \eta^{cf_u}$. We can prove that the sub-formula captures the semantics of cf_u when all the Constraints of cf_u's Operands evaluate to *False*. The proof follows the one for base case. (2)$\Phi^{cf_u} = \Psi^{cf_u} \wedge \Phi^{cf_v}$. We wish to prove that the sub-formula captures the semantics of cf_u if not all the Constraints of cf_u's Operands evaluate to *False*, and the Operator of cf_u is not alt or $loop$. With our assumption, Φ^{cf_v} still captures the semantics of cf_v. $\Psi^{cf_u} = \tilde{\theta}^{cf_u} \wedge \bigwedge\limits_{i \in LN(cf_u)} \tilde{\gamma}_i^{cf_u}$. $\tilde{\theta}^{cf_u}$ captures the order of OSs directly enclosed in cf_u, while $\bigwedge\limits_{i \in LN(cf_u)} \tilde{\gamma}_i^{cf_u}$ captures the order between cf_u and its preceding/succeeding Interaction Fragments. The proof follows the one for base case. The semantics of the OSs directly enclosed in cf_u and the semantics of cf_v are independent. Therefore Ψ^{cf_u} and Φ^{cf_v} are connected using conjunction. In this way, we can prove that Φ^{cf_u} captures the semantics of cf_u. (3)$\Phi^{cf_u} = \Psi_{alt}^{cf_u}$ or $\Phi^{cf_u} = \Psi_{loop}^{cf_u}$ respectively. Similarly, we can prove that the sub-formula captures the semantics of cf_u if not all the Constraints of cf_u's Operands evaluate to *False*, and the Operator of cf_u is alt or $loop$.

Finally, we discuss the sub-formula $\varepsilon_{seq_{n+1}^{nested}}$. It represents that only one OS in $|AOS(seq_{n+1}^{nested})|$ executes at a time, or all OSs in $|AOS(seq_{n+1}^{nested})|$ have executed. Function $AOS(seq_{n+1}^{nested})$ returns the set of chosen and enabled OSs within seq_{n+1}^{nested}, where $|AOS(seq_{n+1}^{nested})| = 2h + 2p_{n+1}$. From lemma 4.10, if $\sigma' \models \varepsilon_{seq_{n+1}^{nested}}$, then $\sigma = \sigma_{[1..2h+2p_{n+1}]} \cdot \tau^{\omega}$. Therefore, $\varepsilon_{seq_{n+1}^{nested}}$ captures the interleaving semantics of seq_{n+1}^{nested}.

Now we have proven that $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}^{nested}})^{\omega}$, respects all the semantic aspects of seq_{n+1}^{nested}, *i.e.*, $\sigma'_{[1..2h+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$.

To conclude, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$, then $\upsilon' \cdot \tau^{\omega} \models \widetilde{\Pi}_{seq_{n+1}^{nested}}$, and $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}^{nested}})^{\omega}$, then $\sigma'_{[1..2h+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}^{nested}})^*$. $\square$

B.3 Proof of Theorem 6.14

Theorem 6.14. For a given Sequence Diagram, seq, with j Messages, $(\Sigma_{sem}^{seq})^*$ and $PRE_{2j}((\Sigma_{NuSMV}^{seq})^{\omega})$ are equal.

Proof. We use mathematical induction, which is based on the number of Messages, j, within seq.

Base step. Basic Sequence Diagram seq_1 contains only one Message, m_1. ($j = 1$)

- Case 1. Sending OS s_1, and receiving OS r_1 of Message m_1 locate on two Lifelines L_1, L_2 respectively (see figure B.1).

  $\Sigma_{sem}^{seq_1} = \{s_1, r_1\}$, where $\Sigma_{sem}^{seq_1} \subseteq \Sigma$. The semantic aspects of seq_1 define that, for m_1, r_1 can only happen after s_1. Only one trace, $v =< s_1, r_1 >$ of size 2, can be derived from seq_1, *i.e.*, $(\Sigma_{sem}^{seq_1})^* = \{< s_1, r_1 >\}$.

  We wish to prove that $< s_1, r_1 > \cdot\tau^\omega \in (\Sigma_{NuSMV}^{seq})^\omega$. The NuSMV model for seq_1 is shown in figure B.8.

  In the Lifeline modules, each variable of OS can become to *True* only once, which means each OS can execute once and only once. OS r_1 takes the state on L_1 as an enabling condition, which means r_1 can be enabled to execute if s_1 on L_1 has executed. In the main module, the INVAR statement restricts that at most one Lifeline can execute an OS at a time. $< s_1, r_1 > \cdot\tau^\omega$ satisfies these restrictions of M_{seq_1} because (1)s_1 and r_1 occur once and only once; (2)s_1 happens before r_1; and (3)s_1 and r_1 do not happen at the same state. Thus, $< s_1, r_1 > \cdot\tau^\omega \models\in (\Sigma_{NuSMV}^{seq})^\omega$.

  We wish to prove that $\forall\sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, then $\sigma_{[1..2]} \in (\Sigma_{sem}^{seq_1})^*$.

  The INVAR statement in the main module restricts that s_1 and r_1 do not happen at the same time. Thus, $\sigma_{[1..2]}$ can be $< s_1, s_1 >, < r_1, r_1 >, < s_1, r_1 >$ or $< r_1, s_1 >$. The variables of OSs in Lifeline modules define that s_1 and r_1 can occur once and only once respectively. Therefore, $\sigma_{[1..2]}$ can be $< s_1, r_1 >$ or $< r_1, s_1 >$. OS r_1's enabling condition represents that r_1 cannot happen before s_1. Therefore, $\sigma_{[1..2]}$ can only be $< s_1, r_1 >$, which is an element of $(\Sigma_{sem}^{seq_1})^*$. In this way, we can prove $\sigma_{[1..2]} \in (\Sigma_{sem}^{seq_1})^*$.

- Case 2. Sending OS s_1, and receiving OS r_1 of Message m_1 locate on a single Lifeline L_1 (see figure B.2).

```
MODULE main
 VAR
  l_L1:  L1(l_L2);
  l_L2:  L2(l_L1);
INVAR
 (((l_L1.chosen -> l_L1.enabled)
  &(l_L2.chosen -> l_L2.enabled))
  &
  ((l_L1.chosen & !l_L2.chosen)
  |(!l_L1.chosen & l_L2.chosen)
  |(!l_L1.enabled & !l_L2.enabled)))

MODULE L1(L2)
 VAR
  OS_s1 : boolean;
  state : {sinit, s1};
  chosen : boolean;
 DEFINE
  s1_enabled := state = sinit;
  enabled := s1_enabled;
  flag_final := state = s1;
 ASSIGN
  init(state) := sinit;
  next(state) :=
   case
    state = sinit & next(OS_s1)    :s1;
    1                              :state;
   esac;
  init(OS_s1) := FALSE;
  next(OS_s1) :=
   case
    chosen & s1_enabled :TRUE;
    OS_s1               :FALSE;
    1                   :OS_s1;
   esac;

MODULE L2(L1)
 VAR
  OS_r1 : boolean;
  state : {sinit, r1};
  chosen : boolean;
 DEFINE
  r1_enabled := state = sinit & L1.state = s1;
  enabled := r1_enabled;
  flag_final := state = r1;
 ASSIGN
  init(state) := sinit;
  next(state) :=
```

```
  case
   state = sinit & next(OS_r1)    :r1;
   1                              :state;
  esac;
 init(OS_r1) := FALSE;
 next(OS_r1) :=
  case
   chosen & r1_enabled :TRUE;
   OS_r1               :FALSE;
   1                   :OS_r1;
  esac;
```

Figure B.8: seq_1 to NuSMV (case 1)

Besides the semantic aspects discussed in case 1, the OSs on L_1 respect their graphical order, i.e., s_1 occurs before r_1. Trace $\upsilon =< s_1, r_1 >$ of size 2 can be derived from seq_1, i.e., $(\Sigma_{sem}^{seq_1})^* = \{< s_1, r_1 >\}$.

The NuSMV model for seq_1 is shown in figure B.9

Comparing to M_{seq_1} in case 1, both OSs are defined in module L_1. The OSs can still only happen once, and s_1 occurs before r_1. Trace $< s_1, r_1 > \cdot \tau^\omega$ can be generated from the NuSMV model, i.e., $(\Sigma_{NuSMV}^{seq_1})^\omega = \{< s_1, r_1 > \cdot \tau^\omega\}$.

Similarly, we wish to prove that $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq})^\omega$; and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, then $\sigma_{[1..2]} \in (\Sigma_{sem}^{seq_1})^*$. The proof follows the one of case 1.

To sum up, for a basic Sequence Diagram with one Message, $(\Sigma_{sem}^{seq})^*$ and $pre((\Sigma_{NuSMV}^{seq})^\omega)$ are equal.

Inductive step. Basic Sequence Diagram seq_n contains n Messages, which are graphically-ordered, i.e., (m_{i-1} locates above m_i $(2 \le i \le k)$). The Messages have $2n$ OSs, which locate on k Lifelines. We assume $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_n})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_n})^\omega$; and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_n})^\omega$, then $\sigma_{[1..2n]} \in (\Sigma_{sem}^{seq_n})^*$ $(j = n)$.

We add a Message, m_{n+1}, at the bottom of seq_n graphically to form a new Sequence Diagram,

```
MODULE main
 VAR
  l_L1:  L1;
INVAR
 ((l_L1.chosen -> l_L1.enabled)
 &(l_L1.chosen |!l_L1.enabled))

MODULE L1
 VAR
  OS_s1 : boolean;
  OS_r1 : boolean;
  state : {sinit, s1, r1};
  chosen : boolean;
 DEFINE
  s1_enabled := state = sinit;
  r1_enabled := state = s1;
  enabled := s1_enabled | r1_enabled;
  flag_final := state = r1;
 ASSIGN
  init(state) := sinit;
  next(state) :=
   case
    state = sinit & next(OS_s1)    :s1;
    state = s1 & next(OS_r1)       :r1;
    1                              :state;
   esac;
  init(OS_s1) := FALSE;
  next(OS_s1) :=
   case
    chosen & s1_enabled :TRUE;
    OS_s1               :FALSE;
    1                   :OS_s1;
   esac;
  init(OS_r1) := FALSE;
  next(OS_r1) :=
   case
    chosen & r1_enabled :TRUE;
    OS_r1               :FALSE;
    1                   :OS_r1;
   esac;
```

Figure B.9: seq_1 to NuSMV (case 2)

seq_{n+1}, with $n+1$ Messages. We wish to prove $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$; and $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$ ($j = n+1$).

(a) We wish to prove $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$.

The semantic aspects of seq_{n+1} enforce that only one OS occurs at a time, and each OS happens once and only once. $\Sigma_{sem}^{seq_{n+1}} = \Sigma_{sem}^{seq_n} \cup \{s_{n+1}, r_{n+1}\}$, where $|\Sigma_{sem}^{seq_n}| = 2n$ and $|\Sigma_{sem}^{seq_{n+1}}| = 2n+2$. If $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then υ' is a finite trace of size $2n+2$, which contains OSs in $\Sigma_{sem}^{seq_{n+1}}$. Adding m_{n+1} at the bottom of seq_n does not change the structure of seq_n. Thus, for trace υ', the order of OSs in $\Sigma_{sem}^{seq_n}$ is still preserved. Message m_{n+1} restricts that s_{n+1} must happen before r_{n+1}, i.e., s_{n+1} locates before r_{n+1} in υ'.

When we modify the NuSMV model for seq_n (M_{seq_n}) to the NuSMV model for seq_{n+1} ($M_{seq_{n+1}}$), we need to add variables and derived variables of s_{n+1} and r_{n+1} in the modules of the Lifelines where these new OSs are located. Accordingly, we need to modify the variable $state$ in these Lifeline modules to record the execution of the new OSs. If new OSs locate on the Lifelines not in seq_n, we also need to change the INVAR statement in the main module to include the new Lifelines into the interleaving semantics. In order to prove $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, we wish to prove that $\upsilon' \cdot \tau^\omega$ satisfies all the restrictions defined by $M_{seq_{n+1}}$, *i.e.*, the restrictions defined by M_{seq_n}, the restrictions defined by variables of s_{n+1} and r_{n+1}, and the restrictions defined by modifying variable $state$ and INVAR statement. With assumption, we know $\upsilon \cdot \tau^\omega$ satisfies all the restrictions defined by M_{seq_n}. As we discussed, the order of OSs within seq_n is still preserved in υ'. Thus, $\upsilon' \cdot \tau^\omega$ also satisfies all the restrictions defined by M_{seq_n}. The variables and derived variables of s_{n+1} and r_{n+1} in $M_{seq_{n+1}}$ define that s_{n+1} and r_{n+1} can occur once and only once, and s_{n+1} must happen before r_{n+1}. $\upsilon' \cdot \tau^\omega$ satisfies these constraints introduced by the new OSs because (1) only one s_{n+1} and one r_{n+1} are in υ', and (2) s_{n+1} locates before r_{n+1} in υ'. The restrictions introduced by modifying variable $state$ of these Lifelines where the new OSs are located and INVAR statement may be various depending on the locations of the new OSs. We discuss the location of the new OSs using four cases as below.

- Case 1: Two OSs of m_{n+1} locate on two new Lifelines, L_{k+1} and L_{k+2} (see figure B.4a); or two OSs of m_{n+1} locate on one new Lifeline, L_{k+1} (see figure B.4b).

  In $M_{seq_{n+1}}$, we add two Lifeline modules for L_{k+1} and L_{k+2} (or one Lifeline module for L_{k+1}). s_{n+1} is defined as a variable in the module for L_{k+1}, while r_{n+1} is defined as a variable in the module for L_{k+2} (the module for L_{k+1}). r_{n+1} takes the enabling condition that L_{k+2} (or L_{k+1}) should reach the state indicating s_{n+1} has occurred, *i.e.*, r_{n+1} cannot happen until s_{n+1} has occurred. No order between s_{n+1}, r_{n+1} and other OSs within seq_{n+1} are enforced by $M_{seq_{n+1}}$. In the main module, the INVAR statement is changed to show the interleaving semantics of all $k+2$ (or $k+1$) Lifeline modules, *i.e.*, one of enabled Lifeline modules can execute or no Lifeline modules are enabled.

  In trace $\upsilon' \in \Sigma_{sem}^{seq_{n+1}})^*$, no two OSs can happen at the same time, which satisfies the restriction imposed by INVAR statement. The OSs of m_{n+1} locate on one or two new Lifelines, so m_{n+1} and the existing Messages, $m_1, m_2...m_n$, are interleaved. Therefore, in υ', s_{n+1} or r_{n+1} can locate (1) between any two OSs of seq_n, or (2) before all OSs of seq_n, or (3) after all OSs of seq_n. Hence, s_{n+1} can be the sth OS of υ', where $1 \leq s \leq 2n+1$; and r_{n+1} can be the rth OS of υ', where $s < r \leq 2n+2$. Therefore, υ' satisfies all the restrictions of $M_{seq_{n+1}}$.

- Case 2: Sending OS s_{n+1} locates on a new Lifeline, L_{k+1}, and receiving OS r_{n+1} locates on an existing Lifeline, L_i $(1 \leq i \leq k)$ (see figure B.4c).

  In M_{seq_n}, we assume the last variable for OS in Lifeline module for L_i is the variable for OS_{pre}. In $M_{seq_{n+1}}$, we add a variable for r_{n+1} in the module for L_i. We also add one Lifeline module for L_{k+1}, which contains a variable for s_{n+1}. r_{n+1} takes two enabling conditions (1)$state$ sets to OS_{pre} to indicate that OS_{pre} has executed; (2)L_{k+1} should reach the state indicating s_{n+1} has occurred. In the main module, the INVAR statement is changed to show the interleaving semantics of all $k+1$ Lifeline modules, *i.e.*, one of enabled Lifeline modules can execute or no Lifeline modules are enabled.

We add m_{n+1} at the bottom of seq_n to form seq_{n+1}, where r_{n+1} becomes the last OS on L_i instead of OS_{pre}. Therefore, OS_{pre} should happen before r_{n+1}. s_{n+1} locates on a new Lifeline, so it is interleaved with the OSs of seq_n. However, s_{n+1} must happen before r_{n+1}. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre} is the pth OS, where $1 \leq p \leq 2n+1$. Then s_{n+1} is the sth OS of υ', where $1 \leq s \leq 2n+1$ and $s \neq p$; r_{n+1} is the rth OS of υ', where $s < r \leq 2n+2$ and $p < r \leq 2n+2$. Thus, υ' satisfies the restrictions imposed by variables for s_1 and r_1. In υ', no two OSs can happen at the same time, which satisfies the restriction imposed by INVAR statement. Therefore, υ' satisfies all the restrictions of $M_{seq_{n+1}}$.

- Case 3: Sending OS s_{n+1} locates on an existing Lifeline, L_i $(1 \leq i \leq k)$, and receiving OS r_{n+1} locates on a new Lifeline, L_{k+1} (see figure B.4d); or two OSs of m_{n+1} locate on an existing Lifeline L_i $(1 \leq i \leq k)$ (see figure B.4e).

  Similarly, in M_{seq_n}, we assume the last variable for OS in Lifeline module for L_i is the variable for OS_{pre}. In $M_{seq_{n+1}}$, we add a variable for s_{n+1} in the module for L_i. We also add one Lifeline module for L_{k+1}, which contains a variable for r_{n+1} (or the variable for r_{n+1} is in the module for L_i). s_{n+1} takes an enabling conditions that $state$ sets to OS_{pre} indicating OS_{pre} has executed. r_{n+1} takes an enabling conditions that L_i should reach the state indicating s_{n+1} has occurred. In the main module, the INVAR statement is changed to show the interleaving semantics of all $k+1$ Lifeline modules (or keep unchanged for k Lifelines if no Lifeline is added).

  We add m_{n+1} at the bottom of seq_n to form seq_{n+1}, where s_{n+1} becomes the OS locating below OS_{pre} on L_i. Therefore, OS_{pre} should happen before s_{n+1}. r_{n+1} cannot happen before s_{n+1} finishes execution. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre} is the pth OS, where $1 \leq p \leq 2n+1$. Then s_{n+1} is the sth OS of υ', where $1 \leq s \leq 2n+1$ and $s \neq p$; r_{n+1} is the rth OS of υ', where $s < r \leq 2n+2$ and $p < r \leq 2n+2$. Thus, υ' satisfies the restrictions imposed by variables for s_1 and r_1. In υ', no two OSs can happen at the same time, which satisfies the restriction imposed by INVAR statement. Therefore, υ' satisfies all the restrictions of $M_{seq_{n+1}}$.

- Case 4: Two OSs of m_{n+1} locate on two existing Lifelines. Without loss of generality, we assume that sending OS s_{n+1} locates on Lifeline L_i $(1 \leq i \leq k)$, receiving OS r_{n+1} locates on Lifeline L_j $(1 \leq j \leq k)$ (see figure B.4f).

  In M_{seq_n}, we assume the last variable for OS in Lifeline module for L_i is the variable for OS_{pre_s}, and the last variable for OS in Lifeline module for L_j is the variable for OS_{pre_r}. In $M_{seq_{n+1}}$, we add a variable for s_{n+1} in the module for L_i, and a variable for r_1 in the module for L_j. s_{n+1} takes an enabling conditions that $state$ sets to OS_{pre_s} indicating OS_{pre_s} has executed. r_{n+1} takes two enabling conditions (1)$state$ sets to OS_{pre_r} to indicate that OS_{pre_r} has executed; (2)L_i should reach the state indicating s_{n+1} has occurred. The INVAR statement in the main module is not changed.

  Adding m_{n+1} at the bottom of seq_n makes that s_{n+1} becomes the OS locating below OS_{pre_s} on L_i, and r_{n+1} becomes the OS locating below OS_{pre_r} on L_j. Therefore, OS_{pre_s} should happen before s_{n+1}, while OS_{pre_r} should happen before r_{n+1}. In trace $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, if OS_{pre_s} is the p_sth OS, where $1 \leq p_s \leq 2n$, and OS_{pre_r} is the p_rth OS, where $1 \leq p_r \leq 2n+1$. Then s_{n+1} is the sth OS of υ', where $p_s < s \leq 2n+1$; r_{n+1} is the rth OS of υ', where $p_r < r \leq 2n+2$. Therefore, υ' satisfies all the restrictions of $M_{seq_{n+1}}$.

To conclude, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$

(b) We wish to prove $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

We wish to prove that $\upsilon' \cdot \tau^\omega$ satisfies all the restrictions defined by $M_{seq_{n+1}}$,

We modify M_{seq_n} to $M_{seq_{n+1}}$ using several steps. (1) Variables and derived variables for s_{n+1} and r_{n+1} are added in the modules of the Lifelines where the OSs are located respectively. (2) Variable $state$ of these Lifeline modules are changed to record the execution of the new OSs. (3) In the main module, the INVAR statement may be changed to represent the interleaving semantics of the existing Lifelines and the new Lifelines. If $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, we wish to prove that $\sigma'_{[1..2n+2]}$

respects all the semantic aspects of seq_{n+1}. We assume that, if $\sigma \in (\Sigma_{NuSMV}^{seq_n})^\omega$, then $\sigma_{[1..2n]}$ respects the semantic aspects of seq_n. The modification of the NuSMV module does not alter the structure of M_{seq_n}, *i.e.*, the order of OSs within seq_n are not changed. Therefore, we can infer that σ' satisfies the semantic aspects of seq_n. In $M_{seq_{n+1}}$, the variables of s_{n+1} and r_{n+1} define that s_{n+1} and r_{n+1} can occur once and only once respectively. r_{n+1} takes an enabling condition indicating s_{n+1} has executed. Thus, σ' respects the semantics of m_{n+1}, *i.e.*, each OS of m_{n+1} happen once and only once, and s_{n+1} must happen before r_{n+1}. We wish to prove that the changes of $state$ in Lifeline modules and INVAR statements in main module make $M_{seq_{n+1}}$ respect the order between the OSs within $\Sigma_{sem}^{seq_n}$ and the OSs of m_{n+1}. which is discussed using four cases as below.

- Case 1: The variables of m_{n+1}'s OSs are added in two new Lifelines modules, the modules for L_{k+1} and L_{k+2}; or the variables of m_{n+1}'s OSs are added in one new Lifeline module, the module for L_{k+1}.

  In seq_{n+1}, we assume that s_{n+1} locates on L_{k+1} and r_{n+1} locates on L_{k+2} or both OSs of m_{n+1} locate on L_{k+1}. The new Message, m_{n+1} and the existing Messages are interleaved. No order among the existing OSs and new OSs are specified in seq_{n+1}. In $M_{seq_{n+1}}$, variables for s_{n+1} and r_{n+1} are added in new Lifeline modules. If $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, then no order among the variables on k Lifelines and the variables on the new Lifelines are restricted in σ'. The INVAR statement is modified to represent the interleaving semantics of all $k+2$ (or $k+1$) Lifeline modules. Therefore, σ' respects the semantic aspect that at most one Lifeline can execute an OS at a time. In this way, σ' respects the semantic aspects of seq_{n+1}.

- Case 2: The variable of s_{n+1} is added in a new Lifeline module, the module for L_{k+1}, and the variable of r_{n+1} is added in an existing Lifeline module, the module for L_i $(i \leq k)$.

  In seq_n, we assume the last OS on L_i is OS_{pre}. In seq_{n+1}, r_{n+1} becomes the last OS on L_i and s_{n+1} locates on L_{k+1}. Therefore, OS_{pre} should happen before r_{n+1}. s_{n+1} is interleaved with the existing OSs. In $M_{seq_{n+1}}$, one Lifeline module for L_{k+1} is added, which contains a variable for s_{n+1}. A variable for r_{n+1} is added in the module for L_i. r_{n+1} takes two enabling conditions. (1)$state$ sets to OS_{pre} to indicate that OS_{pre} has executed; (2)L_{k+1} reaches the state indicating s_{n+1} has occurred. Therefore, in σ', r_{n+1} cannot happen until OS_{pre} and s_{k+1} have executed. Thus, σ' respects the order among the new OSs and the existing OSs defined by seq_{n+1}. The INVAR statement is modified to represent the interleaving semantics of all $k+1$ Lifeline modules. Therefore, σ' respects the semantic aspect that at most one Lifeline can execute an OS at a time. In this way, σ' respects the semantic aspects of seq_{n+1}.

- Case 3: The variable of s_{n+1} is added in an existing Lifeline module, the module for L_i $(i \leq k)$, and the variable of r_{n+1} is added in a new Lifeline module, the module for L_{k+1}; or the variables of both OSs are added in an existing Lifeline module, the module for L_i $(i \leq k)$.

  Similarly, in seq_n, we assume the last OS on L_i is OS_{pre}. In seq_{n+1}, s_{n+1} becomes the OS below OS_{pre} on L_i, and r_{n+1} locates on L_{k+1} (or r_{n+1} locates below s_{n+1} on L_i). Therefore, OS_{pre} should happen before s_{n+1}, and s_{n+1} should happen before r_{n+1}. In $M_{seq_{n+1}}$, a variable for s_{n+1} is added in the module for L_i, taking an enabling condition that $state$ sets to OS_{pre} to indicate OS_{pre} has executed. The variable for r_{n+1} takes an enabling condition that L_i reaches the state indicating s_{n+1} has occurred. Therefore, in σ', s_{n+1} cannot occur until OS_{pre} executes, while r_{n+1} cannot occur until s_{n+1} executes. Thus, σ' respects the order among the new OSs and the existing OSs defined by seq_{n+1}. The INVAR statement is modified to represent the interleaving semantics of all $k+1$ Lifeline modules, or keeps unchanged. Therefore, σ' respects the semantic aspect that at most one Lifeline can execute an OS at a time. In this way, σ' respects the semantic aspects of seq_{n+1}.

- Case 4: The variables of both OSs are added in existing Lifeline modules. Without loss of generality, we assume that the variable of s_{n+1} is added in the module for L_i $(i \leq k)$, and the variable of r_{n+1} is added the module for L_j $(j \leq k)$.

  In seq_n, we assume the last OS on L_i is OS_{pre_s}, while the last OS on L_j is OS_{pre_r}. In seq_{n+1}, s_{n+1} becomes the OS below OS_{pre_s} on L_i, while r_{n+1} becomes the OS below OS_{pre_r} on L_j.

Therefore, OS_{pre_s} should happen before s_{n+1}, and OS_{pre_r} should happen before r_{n+1}. In $M_{seq_{n+1}}$, a variable for s_{n+1} is added in the module for L_i, taking an enabling condition that $state$ sets to OS_{pre_s} to indicate OS_{pre_s} has executed. A variable for r_{n+1} is added in the module for L_j, taking an enabling condition that $state$ sets to OS_{pre_r} to indicate OS_{pre_r} has executed. Therefore, in σ', s_{n+1} cannot occur until OS_{pre_s} executes, while r_{n+1} cannot occur until OS_{pre_r} executes. Thus, σ' respects the order among the new OSs and the existing OSs defined by seq_{n+1}. The INVAR statement keeps unchanged. Therefore, σ' respects the semantic aspect that at most one Lifeline can execute an OS at a time. In this way, σ' respects the semantic aspects of seq_{n+1}.

Now we have proven that $\sigma'_{[1..2n+2]}$ respects all the semantic aspects of seq_{n+1}, i.e., $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

To conclude, $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2]} \in (\Sigma_{sem}^{seq_{n+1}})^*$. $\square$

B.4 Proof of Theorem 6.16

Theorem 6.16. $(\Sigma_{sem}^{seq_r})^*$ and $PRE_{2h+2p}((\Sigma_{NuSMV}^{seq_r})^\omega)$ are equal.

Proof. We use mathematical induction, which is based on the number of CFs, r, directly enclosed in seq_r.

Base step. The sequence Diagram contains at most one CF, cf_1. ($r \leq 1$)

- Case 1. Sequence Diagram seq_0 contains no CF. ($r = 0$)

  The proof follows the one for basic Sequence Diagram.

- Case 2. Sequence Diagram seq_1 contains only one CF, cf_1. ($r = 1$)

  - Case 2.1 We assume that cf_1 has a Operands whose Interaction Constraints evaluate to *False*. The bth Operand contains q_b Messages, where $1 \leq b \leq a$.

    (a) We wish to prove that, $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

We wish to prove that $\upsilon \cdot \tau^{\omega}$ satisfies all the restrictions defined by M_{seq_1}. Similar to the NuSMV model for basic Sequence Diagram, M_{seq_1} still contains a main module and Lifeline modules. Each CEU is declared as a module instance and instantiated in the module of the Lifeline where the CEU locates. A CEU is composed of one or more EUs, each of which is instantiated a module instance inside the CEU module.

First, we consider the restrictions defined by the Lifeline modules. The OSs directly enclosed in the Lifelines are represented as boolean variables. In υ, these OSs respect the semantic rules of seq. It is easy to infer that these OSs also satisfy the restrictions of the Lifeline modules. The proof follows the one for basic Sequence Diagram.

Then, we consider the connection between the OSs and CEUs directly enclosed in each Lifeline. In M_{seq_1}, the CEU module of cf_1 on Lifeline i takes variable $state$ of Lifeline i as an enabling condition, *i.e.*, if $state$ sets to the value indicating that the preceding OS of CEU $cf_i \uparrow_i$ has executed, then the CEU module starts to evaluate the Interaction Constraint locating on the same Lifeline, triggering the execution of the EUs. Therefore, the OSs within a CEU cannot execute until the preceding OS of the CEU finishes execution. If $\upsilon' \cdot \tau^{\omega}$ does not satisfy this restriction, then we assume at least one OS within the CEU, OS_c, occurs before the preceding OS of the CEU, OS_{pre}. The semantic aspects of seq_1 defines that each CF are combined with its preceding OSs using Weak Sequencing. Thus, OS_{pre} must completes execution prior to OS_c's execution, which contradicts our assumption. Therefore, we can prove $\upsilon' \cdot \tau^{\omega}$ satisfies the restriction of the connection between each CEU and its preceding OSs. The CEU's succeeding OS takes variable $flag\_final$ of the CEU module as an enabling condition, which restricts that the succeeding OS cannot execute before the CEU module finishes execution. Similarly, we can prove that $\upsilon' \cdot \tau^{\omega}$ satisfies the restriction of the connection between each CEU and its succeeding OSs.

Finally, we consider restriction defined by the CEU modules. On each Lifeline, the

Interaction Constraints are evaluated when the CEU is ready to execute. Variable op_eva of each Operand takes the value of the Operand's Interaction Constraint to decide if the Operand is enabled to execute. For each EU of the Operand, the variable of the first OS and variable $flag\_final$ take op_eva as a condition. If the Operand is enabled to execute, the first OS of each EU is enabled to execute. Otherwise, the first OS of each EU cannot be enabled to execute and $flag\_final$ evaluates to *True*, indicating that the EU finishes execution. In seq_1, all Operands of cf_1 evaluate to *False*. On each Lifeline, when the preceding OS of cf_1's CEU finishes execution, the CEU reaches its final state to enable its succeeding OS. Therefore, the CEU's preceding OS must happen before its succeeding OS. If υ does not satisfy this restriction, then we assume that, on Lifeline i, the CEU's preceding OS, OS_{pre}, cannot occur until its succeeding OS, OS_{post}, finishes execution. The semantic aspects of seq_1 define that if all Operands of cf_1 evaluate to *False*, then, on each Lifeline, cf_1's preceding OS and succeeding OS are ordered by Weak Sequencing, and cf_1 does not execute. Thus, OS_{pre} must complete execution prior to OS_{post}'s execution, which contradicts our assumption. Therefore, we can prove υ satisfies the restriction defined by the CEU modules. We do not consider the EU modules because they do not execute in this case.

Now we have proven that if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, $\sigma_{[1..2h]} \in (\Sigma_{sem}^{seq_1})^*$.

We wish to prove that $\sigma_{[1..2h]}$ satisfies all the semantic aspects of seq_1. First, we consider the semantic aspect of OSs directly enclosed in seq_1. In M_{seq_1}, variables of the OSs in the Lifeline modules satisfy the restrictions defined by the Lifeline modules. It is easy to infer that these variables also respect the semantic aspect of OSs directly enclosed in seq_1. The proof follows the one for basic Sequence Diagram.

Then, we consider the semantic aspect that cf_1 does not execute because the Constraints of all Operands evaluate to *False*. As we discussed, in M_{seq_1}, variable op_eva

of each Operand takes the value of the Operand's Interaction Constraint to decide if the Operand is enabled to execute. For each EU of the Operand, the variable of its first OS and its variable $flag\_final$ take op_eva as a condition. If the Operand's Constraint evaluate to *True*, the first OS of each EU is enabled to execute. Otherwise, the first OS of each EU is unable to execute and $flag\_final$ evaluates to *True*, indicating that the EU will not execute. In this way, if $\sigma \in (\Sigma^{seq_1}_{NuSMV})^{\omega}$, then $\sigma_{[1..2h]}$ does not contain the OSs within the Operands whose Constraints evaluate to *False*. Therefore, $\sigma_{[1..2h]}$ respects the corresponding semantic aspect of cf_1.

Finally, we consider the semantic aspect that cf_1's preceding OSs and succeeding OSs are connected using Weak Sequencing, *i.e.*, on the same Lifeline, cf_1's preceding OS must happen before its succeeding OS. If $\sigma_{[1..2h]}$ does not respect this semantic aspect, then we assume that, in the module of Lifeline i, variable of the CEU's preceding OS, OS_{pre}, cannot occur until the variable of its succeeding OS, OS_{post}, has executed. Each CEU module takes variable $state$ as a condition to determine when it evaluates the Constraints of its Operands. If $state$ sets to value indicating the CEU's preceding OS has executed, then the Constraints evaluate to *False*, making the CEU reach its final state and the CEU's succeeding OS is enabled to execute. Thus, OS_{pre} must finish execution before OS_{post}, which contradicts our assumption. Therefore, we can prove $\sigma_{[1..2h]}$ respects the semantic aspect of cf_1. We do not consider the semantic aspects of Operands because they do not execute in this case.

Now we have proven that $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma^{seq_1}_{NuSMV})^{\omega}$, respects all the semantic aspects of seq_1, *i.e.*, $\sigma_{[1..2h]} \in (\Sigma^{seq_1}_{sem})^*$.

To conclude, $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma^{seq_1}_{sem})^*$, then $\upsilon \cdot \tau^{\omega} \in (\Sigma^{seq_1}_{NuSMV})^{\omega}$, and $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma^{seq_1}_{NuSMV})^{\omega}$, then $\sigma_{[1..2h]} \in (\Sigma^{seq_1}_{sem})^*$.

– Case 2.2 We assume that cf_1 has at least one Operand whose Constraint evaluates to *True*.

First, we wish to prove that the NuSMV model structure general to all CFs captures the semantics rules general to all CFs.

We assume that, cf_1 has two Operands. One Operand contains p Messages, and its Interaction Constraint evaluates to *True*. The other Operand contains q Messages, and its Interaction Constraint evaluates to *False*. (see figure B.5, where $cond1$ evaluates to *True*, and $cond2$ evaluates to *False*).

(a) We wish to prove that, $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \models \Pi_{seq_1}$.

First, we consider the restrictions defined by the Lifeline modules. We can infer that, in υ, the OSs directly enclosed in seq_1 satisfy the restrictions of the Lifeline modules. The proof follows the one for basic Sequence Diagram.

Then, we consider the order among the OSs and CEUs directly enclosed in each Lifeline. Each Lifeline module and its CEU module restrict that, the CEU module cannot happen until its preceding OS executes, and its succeeding OS cannot happen until the CEU module finishes execution. We can prove that υ satisfies these restriction. The proof follows the one in case 2.1.

Next, we consider restriction defined by the CEU modules. For each EU module inside the CEU module, if its Interaction Constraint evaluate to *True*, the OSs within the EU can be enabled to execute. Otherwise, the EU module reaches its final state, indicating that no OS within the EU will execute. We can prove that υ satisfies these restriction. The proof follows the one in case 2.1.

Finally, we consider restriction defined by the EU modules. The structure of an EU module is quite similar to the structure of a Lifeline module. An EU module restricts that (1) Each OS (not within EU takes $state$ as an enabling condition, which defines that the OS cannot happen until the previous OS finishes execution. (2) For each Message, its receiving OS takes $state$ of the EU where its sending OS locates as an enabling condition, which defines that its receiving OS cannot happen until its sending

OS executes. (3) The variable of each OS defines that each OS can occur once and only once. We can prove that υ satisfies these restriction. The proof follows the one for basic Sequence Diagram.

Now we have proven that if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^*$.

We wish to prove that $\sigma_{[1..2h+2p]}$ satisfies all the semantic aspects of seq_1. First, we consider the semantic aspect of OSs directly enclosed in seq_1. We can infer that, in σ, variables of OSs in Lifeline modules respect the semantic aspects of OSs directly enclosed in seq_1. The proof follow the one in case 2.1.

Then, we consider the semantic aspect that cf_1's preceding/succeeding OSs are combined with cf_1 using Weak Sequencing, *i.e.*, on the same Lifeline, cf_1's preceding OS must happen before its CEU, and cf_1's CEU must happen before its succeeding OS. If $\sigma_{[1..2h+2p]}$ does not respect the semantic aspect between cf_1 and its preceding OSs, then we assume that, in the module of Lifeline i, variable of the CEU's preceding OS, OS_{pre}, cannot occur until the variable of an OS within the CEU, OS_c, has executed. Each CEU module takes variable $state$ as a condition to determine when it evaluates the Constraints of its Operands. If $state$ sets to value indicating the CEU's preceding OS has executed, then the Constraints may evaluate to *True*, enabling the OSs within the CEU to execute. Thus, OS_{pre} must finish execution before OS_c, which contradicts our assumption. Therefore, we can prove that $\sigma_{[1..2h+2p]}$ respects the semantic aspect between cf_1 and its preceding OSs. Similarly, we can prove that $\sigma_{[1..2h+2p]}$ respects the semantic aspect between cf_1 and its succeeding OSs.

Next, we consider the semantic aspect that cf_1's Operands whose Constraints evaluate to *True* can execute, while cf_1's Operands whose Constraints evaluate to *False* are excluded. We can prove that $\sigma_{[1..2h+2p]}$ does not contain the OSs within the Operands whose Constraints evaluate to *False*. Therefore, $\sigma_{[1..2h+2p]}$ respects the semantic aspect. The proof follows the one in case 2.2.

Finally, we consider the semantic aspect that the order of the OSs within each Operand whose Constraint evaluates to *True* is maintained. The order of the OSs within each Operand is similar to the order of the OSs directly enclosed in seq. The order restricts that (1) on a single Lifeline, the OSs respect their graphical order; (2) for a Message, its receiving OS cannot happen until its sending OS executes; (3) each OS executes once and only once. We can prove that $\sigma_{[1..2h+2p]}$ respects these semantic aspects. The proof follows the one for basic Sequence Diagram.

Now we have proven that $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^{\omega}$, respects all the semantic aspects of seq_1, *i.e.*, $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^{*}$.

If cf_1 contains more than two Operands, p Messages may be enclosed in multiple Operands whose Interaction Constraints evaluate to *True*, and q Messages may be enclosed in multiple Operands whose Interaction Constraints evaluate to *False*. The proof follows the one for cf_1 with two Operands.

To conclude, $\forall \upsilon.\upsilon \in \Sigma^{*}$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^{*}$, then $\upsilon \cdot \tau^{\omega} \models \tilde{\Pi}_{seq_1}$, and $\forall \sigma.\sigma \in \Sigma^{\omega}$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^{\omega}$, then $\sigma_{[1..2h+2p]} \in (\Sigma_{sem}^{seq_1})^{*}$.

We have proven the semantic rules general to all CFs can be captured by the NuSMV model general to all CFs. The semantic rules for each CF with different Operator can be enforced by adding different semantic constraints, which are also captured by our NuSMV models. We use Parallel, Alternatives as examples to prove that the semantic rules for each Operator can be captured by the NuSMV model. The cases for CFs with other Operators can be proven similarly.

* Case 2.2.1 We assume that, a given Parallel, cf_1^{par}, has two Operands whose Interaction Constraints evaluate to *True*. The first Operand contains p_1 Messages, and the second Operand contains p_2 Messages. cf_1^{par} covers i Lifelines.

  (a) We wish to prove that, $\forall \upsilon.\upsilon \in \Sigma^{*}$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^{*}$, then $\upsilon \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq_1})^{\omega}$.

The Parallel imposes an interleaving semantics among its Operands. In M_{seq_1}, a boolean variable, $chosen$, is introduced for each EU module to indicate if the EU is chosen to execute. In the main module, an INVAR statement is added for each CEU to restrict that (1) only one enabled EU is chosen to execute an OS; and (2) no EUs are enabled. In υ, only one OS within cf_1 can happen at a time until cf_1 finishes execution. Therefore, we can prove that $\upsilon \cdot \tau^\omega$ satisfies the restriction defined by the Parallel.

We have proven that $\upsilon \cdot \tau^\omega$ satisfies other general restrictions defined by M_{seq_1} in case 2.2. Hence, we can prove that $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, $\sigma_{[1..2h+2p_1+2p_2]} \in (\Sigma_{sem}^{seq_1})^*$.

The semantic aspect of Parallel defines the concurrency among its Operands, *i.e.*, the OSs within the an Operand maintain their order, while the OSs of different Operands are interleaved. If $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, the order of OSs within an Operand is restricted by the EU modules as the general model. The EUs modules inside a CEU module are interleaved, which is restricted by the INVAR statements in the main module. Therefore, we can prove that $\sigma_{[1..2h+2p_1+2p_2]}$ respects the semantic aspect of the Parallel.

We have proven that $\sigma_{[1..2h+2p_1+2p_2]}$ respects other general semantic aspects of seq_1 in case 2.2. Hence, we can prove that $\sigma_{[1..2h+2p_1+2p_2]} \in (\Sigma_{sem}^{seq_1})^*$.

To conclude, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$, and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, then $\sigma_{[1..2h+2p_1+2p_2]} \in (\Sigma_{sem}^{seq_1})^*$.

$*$ Case 2.2.2 We assume that, a given Alternatives, cf_1^{alt}, has two Operands whose Interaction Constraints evaluate to *True*. The first Operand contains p_1 Messages, and the second Operand contains p_2 Messages. cf_1^{alt} covers i Lifelines.

(a) We wish to prove that, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

The semantics of Alternatives defines that at most one of its Operands whose

Constraints evaluate to *True* is chosen to execute. The Operands whose Constraints evaluate to *False* are still excluded. Thus, υ only defines the order of the OSs within the chosen Operand. In M_{seq_1}, a boolean variable, exe, is introduced for each Operand to indicate whether the Operand is chosen to execute. Variable op_eva takes exe as a condition, representing that an Operand can execute if and only if its exe evaluates to *True*. An INVAR statement is added in the main module, indicating that only one Operand's exe can set to *True*, or none of the Constraints of the Operands evaluate to *True*. Thus, M_{seq_1} only restricts the order of OSs within the chosen Operand. Therefore, we can infer that, $\upsilon \cdot \tau^\omega$ satisfies the restriction defined by the Alternatives.

We have proven that $\upsilon \cdot \tau^\omega$ satisfies other general restrictions defined by M_{seq_1} in case 2.2. Hence, we can prove that $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$.

(b) We wish to prove that, $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, $\sigma_{[1..2h+2p_m]} \in (\Sigma_{sem}^{seq_1})^*$ (m is the chosen Operand of cf_1^{alt}).

M_{seq_1} restricts that only EUs of the Operand whose exe evaluates to *True* can be enabled to execute. The INVAR statement of exe restricts that only one exe evaluates to *True* or none of the Constraints evaluate to *True*. The order of the OSs within the chosen Operand is still restricted as the general model. If $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, we can infer that $\sigma_{[1..2h+2p_m]}$ respects the semantics of Alternatives, *i.e.* at most one of its Operands whose Constraints evaluate to *True* is chosen to execute, where m is the chosen Operand.

We have proven that $\sigma_{[1..2h+2p_m]}$ respects other general semantic aspects of seq_1 in case 2.2. Hence, we can prove that $\sigma_{[1..2h+2p_m]} \in (\Sigma_{sem}^{seq_1})^*$.

To conclude, $\forall \upsilon . \upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_1})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_1})^\omega$, and $\forall \sigma . \sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_1})^\omega$, then $\sigma_{[1..2h+2p_m]} \in (\Sigma_{sem}^{seq_1})^*$ (m is the chosen Operand of cf_1^{alt}).

Inductive step. A given Sequence Diagram, seq_n, directly contains n CFs. For the Messages within the CFs, p_n Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We assume $\forall \upsilon.\upsilon \in \Sigma^*$, if $\upsilon \in (\Sigma_{sem}^{seq_n})^*$, then $\upsilon \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_n})^\omega$. $\forall \sigma.\sigma \in \Sigma^\omega$, if $\sigma \in (\Sigma_{NuSMV}^{seq_n})^\omega$, then $\sigma_{[1..2h+2p_n]} \in (\Sigma_{sem}^{seq})^*$. $(r = n)$

We add a CF, cf_{n+1}, in seq_n to form a new Sequence Diagram, seq_{n+1}, with $n+1$ CFs. cf_{n+1} is directly enclosed in seq_{n+1}. In seq_{n+1}, p_{n+1} Messages are chosen and enabled in Operands whose Interaction Constraints evaluate to *True*. We wish to prove that, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$. $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

(a) We wish to prove that, $\forall \upsilon'.\upsilon' \in \Sigma^*$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$.

We wish to prove that $\upsilon \cdot \tau^\omega$ satisfies all the restrictions defined by $M_{seq_{n+1}}$. We extend M_{seq_n} to $M_{seq_{n+1}}$ by adding the CEU and EU modules of cf_{n+1}. However, the restrictions defined by M_{seq_n} are not altered. Thus, the restrictions of $M_{seq_{n+1}}$ consists of the restrictions of M_{seq_n}, the restriction of cf_{n+1} and the restriction defined by the connection between cf_{n+1} and its preceding/succeeding Interaction Fragments.

When we add cf_{n+1} in seq_n to form seq_{n+1}, cf_{n+1} does not change the order of the existing Interaction Fragments. Hence, seq_{n+1} also respects the semantic aspects of seq_n. If $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega$ satisfies the the restrictions of M_{seq_n}. We can also prove that $\upsilon' \cdot \tau^\omega$ satisfies the restrictions of M_{seq_n}. The proof follows the one of base case. We need to prove that $\upsilon' \cdot \tau^\omega$ satisfies the restriction defined by the connection between cf_{n+1} and its preceding/succeeding Interaction Fragments. We discuss cf_{n+1}'s preceding Interaction Fragments using two cases.

- Case i. On Lifeline i, if cf_{n+1}'s preceding Interaction Fragments is OS u , then the CEU of cf_{n+1} take $state$ of the Lifeline module as an enabling condition, indicating u has executed. We can prove that $\upsilon' \cdot \tau^\omega$ satisfies this restriction. The proof follows the one of base case.

- Case ii. If cf_{n+1}'s preceding Interaction Fragments is CF v, then on Lifeline i, the CEU of cf_{n+1} takes variable $flag\_final$ of CEU $v \uparrow_i$ as an enabling condition, *i.e.*, OSs within CEU $cf_{n+1} \uparrow_i$ cannot happen until CEU $v \uparrow_i$ finishes execution. If $\upsilon' \cdot \tau^\omega$ does not satisfy this restriction, then we assume at least one OS within the $cf_{n+1} \uparrow_i$, OS_c, occurs before an OS within $v \uparrow_i$, OS_{pre}. The semantic aspects of seq_{n+1} defines that cf_{n+1} and its preceding Interaction Fragment are combined using Weak Sequencing. Thus, OS_{pre} must completes execution prior to OS_c's execution, which contradicts our assumption. Therefore, we can prove $\upsilon' \cdot \tau^\omega$ satisfies the restriction.

Similarly, we can prove that $\upsilon' \cdot \tau^\omega$ satisfies the restriction defined by the connection between cf_{n+1} and its succeeding Interaction Fragments. Hence, we have proven that υ' satisfies all the restriction of $M_{seq_{n+1}}$.

Now we have proven that if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^*$, then $\upsilon' \cdot \tau^\omega \in \Sigma_{NuSMV}^{seq_{n+1}}$.

(b) We wish to prove that, $\forall \sigma'.\sigma' \in \Sigma^\omega$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^\omega$, then $\sigma'_{[1..2n+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^*$.

If $\sigma' \in (\Sigma_{LTL}^{seq_{n+1}})^\omega$, then $\sigma' = \sigma_{[1..2h+2p_n+2p_{n+1}]} \cdot \tau^\omega$, which follows Lemma 6.15. We wish to prove that $\sigma'_{[1..2h+2p_n+2p_{n+1}]}$ respects all the semantic aspects of seq_{n+1}. Adding cf_{n+1} does not alter the order of the existing Interaction Fragments. Therefore, the semantic aspects of seq_{n+1} consists of the semantic aspects of seq_n, the semantic aspects of m_{n+1} and the semantic aspects defined by the connection between seq_n and m_{n+1}.

When we extend M_{seq_n} to $M_{seq_{n+1}}$, the restrictions defined by M_{seq_n} keep unchanged. We can deduce that $\sigma'_{[1..2n+2p_n+2p_{n+1}]}$ satisfies the restriction of M_{seq_n}. Therefore, $\sigma'_{[1..2n+2p_n+2p_{n+1}]}$ respects the semantic aspects of seq_n. We can also prove that $\sigma'_{[1..2n+2p_n+2p_{n+1}]}$ respects the semantic aspects of m_{n+1}. The proof follows the one of base case. We need to prove that $\upsilon' \cdot \tau^\omega$ respects the semantic aspects defined by the connection between cf_{n+1} and its preceding/succeeding Interaction Fragments. On each Lifeline, if cf_{n+1}'s preceding Interaction Fragment is an OS, then $M_{seq_{n+1}}$ restricts that the CEU of cf_{n+1} takes the preceding OS as an enabling condition, *i.e.*, the CEU of cf_{n+1} cannot happen until the preceding OS executes. If cf_{n+1}'s preceding Interaction Fragment is CF v, then $M_{seq_{n+1}}$ restricts that the CEU of cf_{n+1} takes variable $flag\_final$ of $v \uparrow_i$ as an enabling condition, *i.e.*, the CEU of cf_{n+1} cannot happen until CEU $v \uparrow_i$ finish execution.

These restrictions are consistent with the semantic aspect of seq_{n+1}, which defines that, on each Lifeline, the CEU of cf_{n+1} must take place after its preceding OS/CEU. Therefore, $\upsilon' \cdot \tau^{\omega}$ respects the order between cf_{n+1} and its preceding Interaction Fragment. Similarly, we can prove that $\upsilon' \cdot \tau^{\omega}$ respects the order between cf_{n+1} and its succeeding Interaction Fragment.

Now we have proven that $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^{\omega}$, respects all the semantic aspects of seq_{n+1}, *i.e.*, $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^{*}$.

To conclude, $\forall \upsilon'.\upsilon' \in \Sigma^{*}$, if $\upsilon' \in (\Sigma_{sem}^{seq_{n+1}})^{*}$, then $\upsilon' \cdot \tau^{\omega} \in (\Sigma_{NuSMV}^{seq_{n+1}})^{\omega}$, and $\forall \sigma'.\sigma' \in \Sigma^{\omega}$, if $\sigma' \in (\Sigma_{NuSMV}^{seq_{n+1}})^{\omega}$, then $\sigma'_{[1..2h+2p_n+2p_{n+1}]} \in (\Sigma_{sem}^{seq_{n+1}})^{*}$. $\square$

The semantic aspects of a Sequence Diagram with nested CFs can also be captured using a NuSMV model. The Sequence Diagram is mapped to a main module while each of its Lifeline is mapped to a Lifeline module. Recall that a CF and its Operands are projected onto each of its covered Lifeline to obtain a CEU and EUs respectively. Each CEU is instantiated as a sub-module in its Lifeline module, while each EU within the CEU is instantiated as a sub-module in the CEU module. If an EU encloses other CEUs, each enclosed CEU is mapped to a sub-module in the EU module. We apply this procedure recursively until all CEUs and EUs are mapped into NuSMV modules. We wish to prove that the NuSMV model captures the semantics of the Sequence Diagram precisely. (1) We have proven that the NuSMV model captures the semantics of the Sequence Diagram with directly enclosed CFs. (2) For nested CEUs, their semantics can be captured using the corresponding CEU modules. The proof follows the one for the CEUs directly enclosed in the Sequence Diagram. (3) For EUs which compose the nested CEUs, their semantics can be captured using the corresponding EU modules. The proof follows the one for the EUs which compose the CEUs directly enclosed in the Sequence Diagram. With this sketch, we can prove that the NuSMV model represents the semantics of a Sequence Diagram with nested CFs.

Appendix C: 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 [55]. 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.

C.1 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 *True* can be selected before the mapping from the Sequence Diagram to LTL formulas, *i.e.*, only the Operands whose Constraints evaluate to *True* are mapped to LTL formulas. The auxiliary functions can avoid evaluating Constraints and be implemented directly, *e.g.*, function *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 adapt 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 *True* and mapped to LTL formulas.

C.2 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 *True* (see formula $\bar{\varepsilon}_{seq}$ in figure C.1). An OS can be enclosed into multiple nested CFs, whose Interaction Constraints are associated with the OS, *e.g.*, $cond_m$ is the conjunction of the Interaction Constraints associated with OS_m. Function $AllOS(seq)$ replaces function $AOS(seq)$ in all formulas, which returns all OSs within Sequence Diagram seq. The formula Φ^{CF} is modified as $\bar{\Phi}^{CF}$, which describes the execution of all CF's Operands. For Operand *m*, if the Lifeline where *m*'s Constraint is located is ready to execute the CEU of 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 *True*, function $\bar{\theta}^m$ is satisfied by the Operand and function $\bar{\Phi}^{CF_k}$ is satisfied by each CF_k nested within *m*. Otherwise, the Constraints of Operands of nested CF CF_k set to *False*, denoting no OS within CF_k can occur.

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

Formula γ_i^{CF} 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 η^{CF}. Both formulas uses the macro $\bigwedge\limits_{OS_q \in s} \neg OS_q \,\widetilde{\mathcal{U}}\, OS_p$ to enforces 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 $\Diamond OS_p \rightarrow (\bigwedge\limits_{OS_q \in s} (\neg OS_q) \,\widetilde{\mathcal{U}}\, OS_p)$, 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 $\bar{\mu}^{CF}$ establishes the order between the first occurring OS and other OSs within the same Operand as we described in section 4.4.3.

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 $\bigwedge\limits_{OS_k \in M} \diamondminus OS_k$ to denote that all OSs within *M* have occurred. Since some OSs may not happen, the sub-formula is modified as $\bigwedge\limits_{OS_k \in M} (\Box \neg OS_k)$, which denotes each OS within *M* have occurred or can not occur any more.

Figure C.1 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.

$$\Pi_{seq} = \bigwedge_{i \in LN(seq)} (\bigwedge_{g \in ABEU(seq\uparrow_i)} \alpha_g) \quad \wedge \quad \bigwedge_{j \in MSG(seq)} \beta_j \quad \wedge \quad \bigwedge_{CF \in Anested(seq)} \bar{\Phi}^{CF} \quad \wedge \quad \bar{\varepsilon}_{seq}$$

$$\bar{\varepsilon}_{seq} = \Box(((\widehat{\bigvee_{OS_m \in AllOS(seq)}} OS_m) \quad \vee \quad (\bigwedge_{OS_m \in AllOS(seq)} (\neg OS_m)) \quad \wedge \quad (\bigwedge_{OS_m \in AllOS(seq)} (OS_m \rightarrow cond_m))))$$

$$\begin{aligned}\bar{\Phi}^{CF} = & \bigwedge_{m \in OPND(CF)} \Box(((\bigwedge_{OS_{pre} \in pre(CF\uparrow_{L_m})} (\Box\neg OS_{pre}) \wedge cond_m) \rightarrow (\bar{\theta}^m \wedge \Box cond_m \wedge \bigwedge_{CF_k \in Anested(m)} \bar{\Phi}^{CF_k})) \\ & \wedge ((\bigwedge_{OS_{pre} \in pre(CF\uparrow_{L_m})} (\Box\neg OS_{pre}) \wedge (\neg cond_m)) \rightarrow (\Box(\neg cond_m) \wedge \bigwedge_{n \in AnestedOP(m)} \Box(\neg cond_n)))) \\ & \wedge \bar{\gamma}^{CF} \wedge \bar{\eta}^{CF} \wedge \bar{\mu}^{CF}\end{aligned}$$

$$\bar{\theta}^m = \bigwedge_{i \in LN(m)} (\bigwedge_{g \in ABEU(m\uparrow_i)} \bar{\alpha}_g) \wedge \bigwedge_{j \in MSG(m)} \bar{\beta}_j$$

$$\begin{aligned}\bar{\alpha}_g = & (\bigwedge_{k \in [r..r+|AOS(g)|-2]} ((\neg OS_{k+1} \, \widetilde{\mathcal{U}} \, OS_k) \vee (\diamondminus OS_k))) \\ & \wedge (\bigwedge_{OS_e \in AOS(g)} ((\neg OS_e \, \widetilde{\mathcal{U}} \, (OS_e \wedge \bigcirc\Box\neg OS_e)) \vee (\neg OS_e \wedge \diamondminus OS_e)))\end{aligned}$$

$$\bar{\beta}_j = (\neg RCV(j) \, \widetilde{\mathcal{U}} \, SND(j)) \vee (\diamondminus SND(j))$$

$$\begin{aligned}\bar{\gamma}^{CF} = & \bigwedge_{i \in LN(CF)} (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\Diamond OS_{pre} \rightarrow ((\bigwedge_{OS \in TAllOS(CF\uparrow_i)} (\neg OS)) \, \widetilde{\mathcal{U}} \, OS_{pre})) \\ & \wedge \bigwedge_{OS \in TAllOS(CF\uparrow_i)} (\Diamond OS \rightarrow ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \, \widetilde{\mathcal{U}} \, OS)))\end{aligned}$$

$$\bar{\eta}^{CF} = \bigwedge_{i \in LN(CF)} (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} (\Diamond OS_{pre} \rightarrow ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \, \widetilde{\mathcal{U}} \, OS_{pre})))$$

$$\bar{\mu}^{CF} = \bigwedge_{m \in OPND(CF)} (\bigwedge_{\substack{OS_p \in Init(m) \\ OS_q \in AllOS(m)}} (\neg OS_q \, \widetilde{\mathcal{W}} \, OS_p))$$

$$\begin{aligned}\bar{\Phi}^{CF}_{alt} = & \bigwedge_{m \in OPND(CF)} \Box((((\bigvee_{m \in OPND(CF)} (\bigwedge_{OS_{pre} \in pre(CF\uparrow_{L_m})} (\Box\neg OS_{pre}))) \wedge exe_m) \rightarrow \\ & (\bar{\theta}^m \wedge \Box exe_m \wedge \Box cond_m \wedge \bigwedge_{CF_k \in Anested(m)} \bar{\Phi}^{CF_k})) \\ & \wedge (((\bigvee_{m \in OPND(CF)} (\bigwedge_{OS_{pre} \in pre(CF\uparrow_{L_m})} (\Box\neg OS_{pre}))) \wedge (\neg exe_m)) \rightarrow \\ & (\Box(\neg exe_m) \wedge \Box(\neg cond_m) \wedge \bigwedge_{n \in AnestedOP(m)} \Box(\neg cond_n))) \\ & \wedge \bar{\gamma}^{CF} \wedge \bar{\eta}^{CF} \wedge \bar{\mu}^{CF} \wedge \vartheta^{CF}\end{aligned}$$

$$\begin{aligned}\vartheta^{CF} = & \Box((((\widehat{\bigvee_{m \in OPND(CF)}} exe_m) \wedge \bigwedge_{m \in OPND(CF)} (exe_m \rightarrow cond_m)) \vee (\bigwedge_{m \in OPND(CF)} (\neg cond_m))) \\ & \wedge \bigwedge_{m \in OPND(CF)} (\bigwedge_{n \in AllOS(m)} (OS_n \rightarrow exe_m)))\end{aligned}$$

$$
\begin{aligned}
\bar{\theta}^m_{critical} &= \bigwedge_{i \in LN(m)} \bar{\delta}_{(AllOS(m\uparrow_i)),(AllOS(seq\uparrow_i) \setminus AllOS(m\uparrow_i)))} \\
\bar{\delta}_{M_1,M_2} &= \Box((\bigvee_{OS_k \in M_1} OS_k) \rightarrow ((\bigwedge_{OS_j \in M_2} (\neg OS_j)) \,\widetilde{\mathcal{U}}\, (\bigwedge_{OS_k \in M_1} (\Box \neg OS_k)))) \\
\bar{\theta}^m_{assert} &= \bigwedge_{i \in LN(m)} \bar{\lambda}^{i,seq}_{(pre(m\uparrow_i),AllOS(m\uparrow_i))} \\
\bar{\lambda}^{i,seq}_{N_1,N_2} &= \Box(\bigwedge_{OS_p \in N_1} (\Box \neg OS_p) \rightarrow ((\bigwedge_{OS_q \in (AllOS(seq\uparrow_i) \setminus N_2)} (\neg OS_q)) \,\widetilde{\mathcal{U}}\, (\bigwedge_{OS_r \in N_2} (\Box \neg OS_r))))
\end{aligned}
$$

Figure C.1: LTL formulas for implementation of templates

BIBLIOGRAPHY

[1] PUBLIC LAW 104-191. *Health Insurance Portability and Accountability Act of 1996*. 104th Congress, 1996.

[2] L. Alawneh, M. Debbabi, F. Hassaine, Y. Jarraya, and A. Soeanu. A unified approach for verification and validation of systems and software engineering models. In *ECBS 2006*, pages 409–418, 2006.

[3] Juliana Kuster-Filipe Alessandra Cavarra. Combining Sequence Diagrams and ocl for liveness. In *Proceedings of the Semantic Foundations of Engineering Design Languages (SFEDL)*, pages 19–38, 2004.

[4] Juliana Kuster-Filipe Alessandra Cavarra. Formalizing liveness-enriched sequence diagrams using ASMs. In *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 62–77, 2004.

[5] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of Message Sequence Charts. *TSE*, 31(9):304–313, 2003.

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

[7] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *CONCUR, LNCS*, volume 1664, pages 114–129, 1999.

[8] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *SCESM*, pages 21–28, 2006.

[9] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Autom Softw Eng*, 14(3):293–340, 2007.

[10] Kevin Beaver and Rebecca Herold. *The Practical Guide to HIPAA Privacy and Security Compliance*. Auerbach Publications, 2003.

[11] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *ICSE*, pages 511–520, 2008.

[12] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. From Live Sequence Charts to state machines and back: A guided tour. 31(12):999–1014, 2005.

[13] Maria Victoria Cengarle. Uml 2.0 interactions: Semantics and refinement. In *Proc. 3rd Int. Wsh. Critical Systems Development with UML*, pages 85–99, 2004.

[14] Maria Victoria Cengarle. Operational semantics of UML 2.0 interactions. Technical report, Technische Universitat Munchen, 2005.

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[16] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[17] Security Standard Council. *Payment Card Industry (PCI) Data Security Standard*, 2.0 edition, 2010.

[18] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Trans on Soft. Eng.*, 31, 2005.

[19] Haitao Dan, Robert M. Hierons, and Steve Counsell. Thread-based analysis of sequence diagrams. In *IFIP International Federation for Information Processing 2007*, 2007.

[20] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. Experiences in the logical specification of the hipaa and glba privacy laws. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, WPES '10, pages 73–82, 2010.

[21] Henry DeYoung, Deepak Garg, Dilsun Kaynar, and Anupam Datta. Logical specification of the GLBA and HIPAA privacy laws. Technical Report CMU-CyLab-10-007, CMU, 2010.

[22] Chris Dimick. Californian sentenced to prison for hipaa violation. http://journal.ahima.org/2010/04/29/californian-sentenced-to-prison-for-hipaa-violation.

[23] Alexander Egyed. Resolving uncertainties during trace analysis. In *FSE*, pages 3–12, New York, NY, USA, 2004. ACM.

[24] Alexander Egyed. Instant consistency checking for the UML. In *ICSE*, pages 381–390, 2006.

[25] Alexander Egyed. Fixing inconsistencies in UML design models. In *ICSE*, pages 292–301, 2007.

[26] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *ASE*, pages 99–108, 2008.

[27] Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. Compositional semantics for UML 2.0 Sequence Diagram using Petri Nets. In *SDL LNCS*, volume 3530, pages 133–148, 2005.

[28] Mohammed Elkoutbi and Rudolf K. Keller. User interface prototyping based on uml scenarios and high-level petri nets. In *Proceedings of the 21st international conference on Application and theory of petri nets*, ICATPN'00, pages 166–186, 2000.

[29] Joao M. Fernandes, Simon Tjell, Jens Baek Jorgensen, and Oscar Ribeiro. Designing tool support for translating use cases and UML 2.0 Sequence Diagrams into a coloured Petri Net. In *International Workshop on Scenarios and State Machines*.

[30] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. *TSE*, 20(8):569–578, 1994.

[31] Radu Grosu and Scott A. Smolka. Safety-liveness semantics for UML 2.0 Sequence Diagrams. In *Int. Conf. on Application of Concurrency to System Design*, pages 6–14, 2005.

[32] Elsa L. Gunter, Anca Muscholl, and Doron Peled. Compositional Message Sequence Charts. In *TACAS*, volume 2031, pages 496–511, 2001.

[33] Youcef Hammal. Branching time semantics for uml 2.0 sequence diagrams. In *Proceedings of the 26th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE'06, pages 259–274, 2006.

[34] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML Sequence Diagrams. *Soft. and Sys. Modeling*, 7(2):237–252, 2008.

[35] Oystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stolen. STAIRS towards formal design with Sequence Diagrams. *Software and Systems Modeling*, 4:355–357, November 2005.

[36] Castejon Martinez Humberto Nicolas. Synthesizing state-machine behaviour from uml collaborations and use case maps. In *Proceedings of the 12th international conference on Model Driven*, SDL'05, pages 339–359, 2005.

[37] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

[38] L. Kloul and J. Kuster-Filipe. From interaction overview diagrams to pepa nets. In *4th Workshop on Process Algebras and Timed Activities*, 2005.

[39] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In *MODELS*, pages 42–51, 2006.

[40] Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. Temporal logic for scenario-based specifications. In *TACAS*, pages 445–460, 2005.

[41] Hillel Kugler, Cory Plock, and Amir Pnueli. Controller synthesis from LSC requirements. In *FASE*, pages 79–93, 2009.

[42] Rahul Kumar, Eric G. Mercer, and Annette Bunker. Improving translation of Live Sequence Charts to temporal logic. *Electron. Notes Theor. Comput. Sci.*, 250(1):137–152, 2009.

[43] Juliana Küster-Filipe. Modelling concurrent interactions. *Theor. Comput. Sci.*, 351(2):203–220, 2006.

[44] Emmanuel Letier and Axel van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Softw. Eng. Notes*, 27:119–128, 2002.

[45] Stefan Leue and Peter B. Ladkin. Implementing and verifying MSC specifications using PROMELA/XSPIN. In *SPIN96*, volume 32 of *DIMACS*, pages 65–89, 1996.

[46] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *SCESM*, pages 5–11, 2006.

[47] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and Makan Pourzandi. Formal verification and validation of UML 2.0 Sequence Diagrams using source and destination of messages. *Electron. Notes Theor. Comput. Sci.*, 254:143–160, 2009.

[48] Thomas Maier and Albert Zundorf. The fujaba statechart synthesis approach. In *in: 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003.

[49] Erkki Makinen and Tarja Systa. Mas - an interactive synthesizer to support behavioral modelling in uml. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 15–24, 2001.

[50] Kevin McCaney. Hackers steal medical records on 181,000 from utah server. http://gcn.com/articles/2012/04/09/utah-hackers-medicaid-chip-medical-recoreds-breached.aspx.

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

[52] Bill Mitchell. Resolving race conditions in asynchronous partial order scenarios. *TSE*, 31(9):767–784, 2005.

[53] Bill Mitchell. Characterizing communication channel deadlocks in sequence diagrams. *TSE*, 34(3):305–320, 2008.

[54] Anca Muscholl and Doron Peled. Deciding properties of Message Sequence Charts. In *the First Int. Conf. on Foundations of Soft. Science and Comp. Structure, LNCS*, volume 1378, pages 226–242, 1998.

[55] Object Management Group. Unified Modelling Language (Superstructure), v2.3, 2010. Internet: www.omg.org.

[56] HHS Press Office. Rite aid agrees to pay $1 million to settle hipaa privacy case. http://www.hhs.gov/news/press/2010pres/07/20100727a.html.

[57] Doron Peled. Specification and verification of Message Sequence Charts. In *FORTE/PSTV*, pages 139–154, 2000.

[58] A. Pnueli. The temporal logic of programs. In *FOCS*, volume 526, pages 46–67, 1977.

[59] Mark Robinson, Hui Shen, and Jianwei Niu. *High Assurance BPEL Process Models*, chapter 11, pages 219–240. 2009.

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

[61] Hui Shen, Ram Krishnan, and Jianwei Niu. Visually analyzing HIPAA using interaction diagram. *IEEE Transactions on Dependable and Secure Computing, to be submitted*, 2013.

[62] Hui Shen, Mark Robinson, and Jianwei Niu. Formal analysis of scenario aggregation. Technical Report CS-TR-2010-03, UTSA, 2010.

[63] Hui Shen, Mark Robinson, and Jianwei Niu. Formal analysis of Sequence Diagram with combined fragments. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT)*, pages 44–54, July 2012.

[64] Hui Shen, A. Virani, and Jianwei Niu. Formalize UML 2 Sequence Diagrams. In *HASE*, pages 437–440, Dec. 2008.

[65] Rocky Slavin, Hui Shen, and Jianwei Niu. Characterizations and boundaries of security requirements patterns. In *Second International Workshop on Requirements Patterns (RePa)*, pages 48–53, September 2012.

[66] Harald Storrle. Trace semantics of interactions in UML 2.0. Technical report, LMU Munchen, 2004.

[67] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *29th International Conference on Software Engineering*, pages 34–43, 2007.

[68] Sebastian Uchitel, Jeff Kramer, and Jeff Magge. Synthesis of behavioral models from scenarios. *TSE*, 29(2):99–115, February 2003.

[69] M.F. van Amstel, C.F.J. Lange, and M.R.V. Chaudron. Four automated approaches to analyze the quality of UML Sequence Diagrams. In *COMPSAC*, volume 2, pages 415–424, 2007.

[70] Axel van Lamsweerde and Laurent Willemet. Inferring declarative requirements specifications from operational scenarios. 24(12):1089–1114, 1998.

[71] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *ASE*, pages 248–257, 2008.

[72] Jon Whittle. Precise specification of use case scenarios. In *FASE*, pages 170–184, 2007.

[73] Jon Whittle and Praveen K. Jayaraman. Synthesizing hierarchical state machines from expressive scenario descriptions. *TOSEM*, 19(3):1–45, 2010.

[74] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000.

[75] Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 242–251, 2004.

VITA

Hui Shen received her B.S. in Computer Science from School of Software Engineering at Beijing Institute of Technology in 2006. After graduation, Hui immediately started pursuing her Ph.D. degree in the Department of Computer Science at the University of Texas at San Antonio in the Fall 2006 semester. She began to work with Dr. Jianwei Niu from 2007. Her research interest is software engineering, focusing on formal methods and requirements engineering, including automated software analysis techniques and tools.