Formal Analysis of Scenario Aggregation

Hui Shen, Mark Robinson and Jianwei Niu

Department of Computer Science
University of Texas at San Antonio
{hshen, mrobinso, niu}@cs.utsa.edu

Technical Report: CS-TR-2010-003

Abstract. 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 UML Sequence Diagram permit various types of control flow among messages (e.g., interleaving and iteration) to express an aggregation of multiple scenarios encompassing very complex and concurrent behaviors. Understanding the behavior of such Sequence Diagrams can be difficult, particularly if the Combined Fragments have semi-formal semantics. We introduce an approach to formalize the semantics of Sequence Diagrams with Combined Fragments in terms of both NuSMV models and Linear Temporal Logic formulas. These two formalizations enable us to leverage the analytical powers of model checking to automatically determine if a collection of Sequence Diagrams is consistent, safe, and adheres to user-supplied properties. Our work increases the accessibility of formal verification techniques to practitioners, allowing them to remain focused in the realm of scenario-based, intuitive specifications.

1 Introduction

The software development community is adopting model-driven engineering as a viable practice to improve the productivity and quality of software systems. Scenario-based models have been widely employed for the description of interactions among environmental actors (e.g., other software packages or human beings) and the components of the software systems. UML Sequence Diagrams, which graphically depict scenarios, have served as well-accepted and intuitive media among software practitioners and tool builders. Combined Fragments are control constructs that increase a Sequence Diagram's expressiveness, allowing multiple scenarios, or traces, within it. However, Combined Fragments increase the difficulty of Sequence Diagram analysis due to the complexity of computing all the partial orders of the sending and receiving of messages.

First, Combined Fragments include non-sequential control flow constructs and can be nested within each other, making it difficult to comprehend what behavior is possible in all traces. Exacerbating this problem, the semantics of Combined Fragments is not formally defined compared to their precise syntax descriptions [20]. Second, errors from concurrency can easily be introduced that are subtle enough to evade discovery via manual inspection or simulation. Third, software designers can construct multiple Sequence Diagrams that are complementary perspectives of a single system. Determining that these Sequence Diagrams provide a consistent description can therefore be extremely difficult.

To address these problems, we introduce an automated technique to facilitate the verification of Sequence Diagrams by leveraging the analytical powers of model checking. We formally describe each Combined Fragment in terms of NuSMV [6] modules. Then the model checking mechanism can explore all possible traces specified in the Sequence Diagram, verifying if user-defined properties are satisfied. Furthermore, we develop a formal template to represent Sequence Diagrams with Combined Fragments as Linear Temporal Logic (LTL) [22] formulas. Our template is comprised of simpler definitions, each of which represents a separate aspect of the Combined Fragment's semantics. Nested Combined Fragments may be easily represented as conjunctions of LTL templates. We believe one can use LTL templates to describe the various semantics of scenario-based languages. While dual formalization may seem redundant, we can evaluate the conformance of the two separate representations, helping us gain higher-level assurance of our transformation processes.

Using LTL templates, we translate Assertion and Negation Combined Fragments, representing mandatory and forbidden behaviors respectively, into LTL specifications to express consistency and safety properties of a system. Thus we can verify that a set of Sequence Diagrams are consistent and safe without requiring users to specify the LTL properties directly. We have developed a tool suite to implement the techniques and generate Sequence Diagram visualizations from NuSMV counterexamples to ease users' effort to locate the property violation. We evaluate our approach by analyzing two design examples taken from an insurance industry software application.

The main contribution of our research is three-fold:

- We show that Sequence Diagrams with Combined Fragments can be captured in two formal notations, as a NuSMV finite state machine (FSM) and LTL, which helps to gain better theoretical understanding of the Sequence Diagram.
- We provide a process for automatic translation of a Sequence Diagram into both notations, allowing users to verify an individual Sequence Diagram and reason about the consistency of a collection of Sequence Diagrams.
- Our approach bridges the gap between Sequence Diagrams and model checking, increasing the accessibility of formal verification techniques to practitioners.

The rest of the paper is structured as follows. Section 2 and 3 summarize the syntax and semantics of Sequence Diagrams. Sections 4 and 5 describe the formal representations of simple Sequence Diagrams and Combined Fragments in terms of NuSMV modules and LTL formulas. Section 6 discusses the generation of the LTL safety and consistency properties from Negative and Assertion Combined Fragments. Section 7 introduces our framework for automated analysis of Sequence Diagrams and evaluates our approach via a case study. Section 8 presents related work. We conclude with Section 9.

2 The UML 2 Sequence Diagram

In this section, we briefly describe the syntax and semantics of a Sequence Diagram with Combined Fragment provided by [20]. Figure 1 shows an example with annotated syntactic constructs.

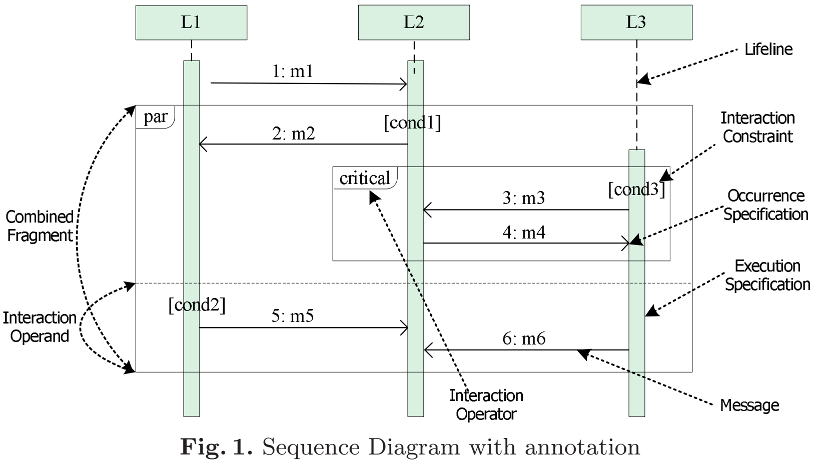

Fig. 1. Sequence Diagram with annotation

2.1 Basic Sequence Diagram Concepts

We refer to a Sequence Diagram without Combined Fragments as a basic Sequence Diagram. A **Lifeline** is a vertical line representing participating object. A horizontal line between Lifelines is a **Message**, which has a name and a (optional) number. Each Message is sent from its source Lifeline to its target Lifeline and has two endpoints. Each endpoint is an intersection with a Lifeline and is called an **Occurrence Specification (OS)**, denoting a sending or receiving event, which is given by $\langle L_i$, msg_j, loc_k, $type\rangle$. L_i denotes its associated Lifeline i, msg_j denotes its associated Message, loc_k is the location where the OS happens on i, and *type* denotes it is either a sending or a receiving event. A finite number of locations are associated for each Lifeline to uniquely express OSs. OSs can also be the beginning or end of an **Execution Specification**, indicating the execution of a unit of behavior within a Lifeline [20].

2.2 Structured Control Constructs

An **Interaction Use** allows one Sequence Diagram to refer to another Sequence Diagram. The referring Sequence Diagram copies the contents of the referenced Sequence Diagram. A **Combined Fragment** (CF) consists of an **Interaction Operator** and one or more **Interaction Operands**. OSs, CFs, and Interaction Operands are collectively called Interaction Fragments. An Interaction Operand may contain a boolean expression which is called an **Interaction Constraint**. An Interaction Constraint is shown in a square bracket covering the Lifeline where the first event will happen. A CF can enclose all, or part of, one or more Lifelines in a Sequence Diagram. The essential CFs that are formalized are:

- **Alternatives**: one of the Operands whose Interaction Constraints evaluate to $True$ is nondeterministically chosen to execute.
- **Option**: its sole Operand executes if the Interaction Constraint is $True$.
- **Break**: its sole Operand executes if 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 Operand will execute for at least the minimum count and no more than the maximum count as long as the Interaction Constraint is $True$.
- **Assertion**: the OSs on a Lifeline within its sole Operand must occur right after the preceding OSs.
- **Negative**: its sole Operand represents the forbidden traces.
- **Weak Sequencing**: on a Lifeline, the OSs within an Operand cannot execute until the OSs in the previous Operand complete, i.e., the execution of the Operands on the same Lifeline adheres to the graphical order.
- **Strict Sequencing**: in any Operand except the first one, OSs cannot execute until the previous Operand completes.
- **Coregion**: the contained Operands are interleaved.

There are other Structured Control Constructs that have been introduced to express various control flows. **General Ordering** imposes a binary relation to restrict the order between two OSs on different Lifelines.

2.3 Semantics for Sequence Diagrams

The semantics of a Sequence Diagram is defined by two sets of traces, one containing a set of valid traces and the other containing a set of invalid traces. The traces in the set of invalid traces are those defined in a Sequence Diagram's Negative CF (NCF). An Assertion CF (ASCF) specifies the mandatory traces in the sense that any trace that is not consistent with them is invalid. Traces specified by a Sequence Diagram are considered as valid traces. Traces that are not specified as either valid or invalid are called inconclusive traces. To provide a commonly accepted and intuitive semantics, we adopt the following well-formed constraints [16]: NCF should be at the top level of a Sequence Diagram. Neither the NCF nor ASCF may be nested within themselves.

A trace is a sequence of OSs representing an end-to-end behavior, expressing Message exchange among multiple Lifelines. The OSs on a single Lifeline without structured control constructs execute in a total order, meaning that the execution adheres to the OSs' graphical order. OSs on different Lifelines are interleaved (only one Lifeline executes an OS in a step), but the receiving OS does not execute until the sending OS has executed. Messages are of two types: synchronous and asynchronous. If a synchronous Message is sent, the source Lifeline is blocked until it receives a response from the target Lifeline [20]. For an asynchronous Message, the source Lifeline can continue to send or receive Messages after the Message is sent. In other words, the sending and receiving OSs of a Message can be interleaved with other OSs. In a Sequence Diagram, the constituent Interaction Fragments are ordered and combined by Weak Sequencing.

3 Formal Semantics of Sequence Diagram

UML Sequence Diagram In this section, we define the operational semantics for Sequence Diagram, which is the first step towards formalizing Sequence Diagrams in terms of NuSMV models and LTL formulas.

3.1 Sequence Diagram Deconstruction

To facilitate the description of CFs in terms of FSMs and LTL formulas, we project every CF cf_k onto each of its covered Lifelines i to obtain a **compositional execution unit (CEU)**, which is denoted by $cf_k \uparrow_i$. (The dotted rectangle on Lifeline $L1$ in figure 1 shows an example). 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 enclosed in the CF on Lifeline L_i. **An execution unit (EU)** is denoted by $op \uparrow_i$. An EU is a **basic execution unit (BEU)** if it does not contain any other EUs. (The dotted rectangles on Lifeline $L2$ in figure 1 shows an example). A BEU u is given by a pair, $\langle \sigma_u[0..n-1], Cond \rangle$, in which $\sigma_u[0..n-1]$ is a finite trace of length n on Lifeline L_i enclosed in Operand op, and $Cond$ is the Interaction Constraint of the Operand. $Cond$ is evaluated at the beginning of the CF execution, and is $True$ when there is no Interaction Constraint. 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.

To represent nested CFs, we define a **hierarchical execution unit (HEU)**, denoted by $\langle setCEU, setBEU, Cond \rangle$, which is an EU containing other CEU(s). For each HEU, $setCEU$ 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. In HEU, the continuous OSs which are directly enclosed in the HEU but not interrupted by any CEU are grouped into a BEU, which inherits its parent HEU's Constraint, $Cond$. The constituent BEU(s) and CEU(s) within an HEU execute sequentially, complying with their graphical order.

Figure 2 show the deconstruction of the Sequence Diagram in figure 1. We consider Lifeline $L2$ as an example to demonstrate the projections of the two CFs. The Parallel CF is projected to obtain

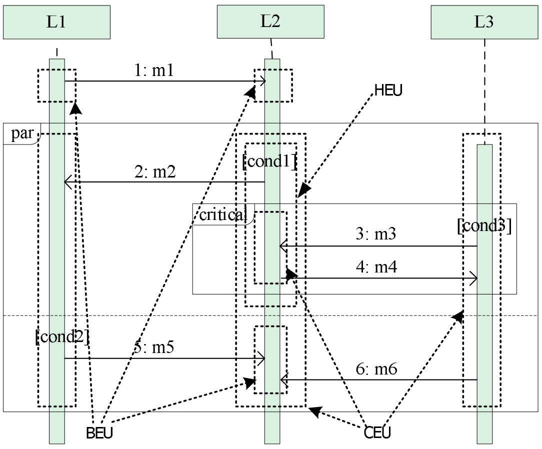

Fig. 2. Sequence Diagram Deconstruction

a CEU. Its first Operand is represented as an HEU, containing the CEU projected from the Critical Region CF. The second Operand of Parallel CF is represented as a BEU. The OS prior to the Parallel CF is grouped into a BEU. We provide a metamodel to show the relations among BEU, HEU and CEU in figure 3.

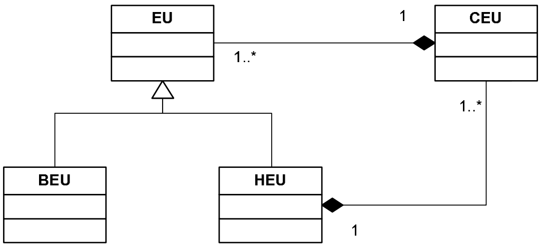

Fig. 3. Execution Unit Metamodel

3.2 Semantics of Basic Sequence Diagram

Two semantic aspects must be considered for the basic Sequence Diagram.

1. On each Lifeline, OSs execute in their graphical order.
2. For the same Message, sending must take place before receiving.

Figure 4 represents an example of a basic Sequence Diagram. On Lifeline $L1$, OS $s3$ can not happen until OS $s1$ executes, which enforces semantic aspect 1. For Message $m1$, OS $r1$ can not happen until OS $s1$ executes, which enforces semantic aspect 2. Sequence Diagram with synchronous Message must obey another semantics rule that if a Lifeline sends a Message, it is blocked until receiving the corresponding reply Message.

3.3 Semantics of CombinedFragments

In the following section, we describe an operational semantics for Interaction Operators of CombinedFragments. The deconstruction of Sequence Diagram allows us to define the semantics of the Interaction Operators in terms of EU and CEU.

We first consider the semantics rules general to all CFs. CF modifies the execution order of OSs on each covered Lifeline by the following semantic rules.

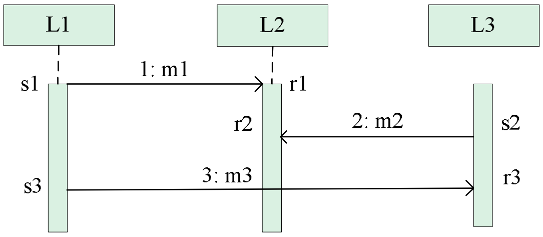

Fig. 4. Basic Sequence Diagram

1. Interaction Fragments, including OSs and CFs, are combined using Weak Sequencing. In other words, CF and its preceding/succeeding Interaction Fragments are ordered by Weak Sequencing.
2. The semantics of each CF Operator (summarized in Section 2) defines the execution of all the Operands. Each Interaction Operator has its specific semantic implications on the execution of events on the covered Lifeline.
3. Within a CF, the order of Interaction Fragments within each Operand are maintained if the constraint of the Operand evaluates to $True$; otherwise, (i.e., the Constraint evaluates $False$,) the CF is excluded. In other words, the CF does not execute when the Constraints of all the Operands evaluate to $False$. Thus, CF's preceding Interaction Fragment and succeeding Interaction Fragment are ordered by Weak Sequencing.

In this paper, we focus on defining the semantics of Sequence Diagrams with simple events, i.e., events without parameters. We assume the Interaction Constraints of CFs are "static", in the sense that they can be evaluated at the beginning of the execution of a Sequence Diagram. Simple events do not change the execution of OSs within Sequence Diagram. In this way, evaluating Constraints at the beginning of execution of the Sequence Diagram is equivalent to evaluating at the beginning of execution of the CFs which enclose them.

These following two semantics rules are common to all CFs.

1. On a single Lifeline, if the conditions of all of the EUs projected from Operands within the CF evaluate to $False$, none of the EUs executes.
2. If an EU is chosen, it continues to execute until completion. Note, the EU may be interleaved by other OSs, which are not directly contained by the CEU in question.

For representation simplicity, we exclude these rule for all CFs.

Alternatives The Alternatives CF (ALCF) 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 ALCF. If none of the Operands has Constraint that evaluate to $True$, none of the EUs are executed and the remainder of the enclosing Interaction Fragment is executed.

On each Lifeline, ALCF makes EUs within a CEU obey following semantics rules:

1. If only one EU's condition evaluates to $True$, the EU is chosen to execute. If more than one EU's condition evaluates to $True$, one EU is nondeterministically chosen to execute. Note, multiple covered Lifeline within the ALCF must select the same Operand to keep the consistency.
2. The chosen EU continues to execute until completion.

Option The Option CF (OCF) represents a choice of behavior that either the (sole) Operand happens or nothing happens. The semantics rules of OCF on a single Lifeline are:

1. The EU executes if its condition evaluates to $True$.
2. The EU continues to execute until completion.

Break The Break CF (BCF) 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. On each Lifeline, we introduce SEU_{post} to denote the reminder of the EU or Sequence Diagram on the Lifeline, that directly encloses the Break CEU, i.e., Break CEU's succeeding sequence of EUs and CEUs within the EU which directly encloses the Break CEU. A BCF can be represented as an ALCF in a straightforward way. We rewrite the semantics interpretation of BCF as an ALCF with two Operands, the Operand of BCF and the Operand representing SEU_{post} on all covered Lifelines. The Constraint of the second Operand is the negation of the first Operand's Constraint. Figure 5 is an example of BCF which is enclosed in a PCF. The reminder of the Operand containing BCF is Message $m4$. We rewrite the Sequence Diagram, using ALCF to replace BCF (see figure 6). $cond3$ is the Constraint of BCF and $cond4$ is the negation of it. In this way, only one Operand can be chosen to execute.

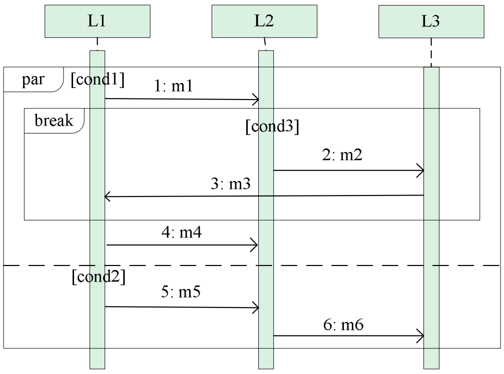

Fig. 5. Example for Break CF

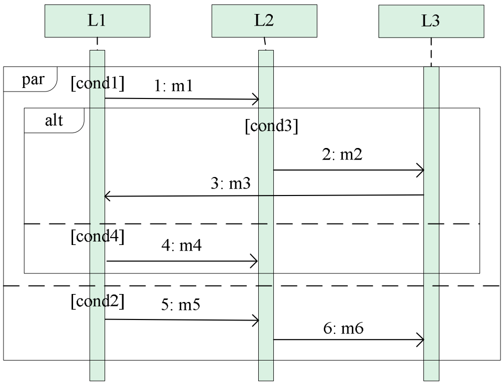

Fig. 6. Example for redraw Break CF to ALCF

The semantics rules of BCF are the same as ALCF with two mutually exclusive Operands as below:

1. If the condition of the break EU evaluates to $True$, the EU executes. Otherwise, SEU_{post} executes.
2. The EU continues to execute until completion.

Collectively, ALCF, OCF and BCF are called branching constructs.

Parallel The Parallel CF (PCF) represents concurrency among its Operands. The OSs of the different Operands can be interleaved as long as the ordering imposed by each Operand is preserved. The semantics rules of PCF on a single Lifeline are provided:

1. If only one EU's condition evaluates to $True$, the EU executes. If more than one condition of EU evaluates to $True$, any EU of them, but only one, executes an OS at a time.
2. The EUs continue to execute until completion in an interleaving manner, i.e, only one EU executes an OS at a time.

Loop The Loop CF (LCF) represents the iterations of the sole Operand. Its condition includes a lower bound, "minint", and an upper bound, "maxint" besides the Constraint of the Operand, which sets restriction on the number of iterations. A loop iterates at least the "minint" number of times and at most the "maxint" number of times. If the Constraint evaluates to false after the minimum number of iterations, the loop will terminate. We represent the semantics rules of LCF on a single Lifeline as:

1. If the number of iteration is less than "minint", the EU executes without checking the condition. If the number of iteration is equal to "minint", the EU executes when the condition evaluates to $True$ or stops execution when the condition evaluates to $False$. When the number of iteration exceeds the "maxint" times, the EU terminates.
2. The EU continues to execute until completion.

Critical Region The Critical Region CF (CRCF) 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.The semantics rules of CRCF on a single Lifeline are:

1. The EU executes if its condition evaluates to $True$.
2. The EU continues to execute until completion restricted for being interleaved by other OSs on the same Lifeline.

Assertion The Assertion CF (ASCF) represents that the behavior described by the Operand is the only valid behavior at that point in the execution. Any other behavior is invalid. The semantics rules of ASCF on a single Lifeline are:

1. The EU executes if its condition evaluates to $True$.
2. The EU is the only valid continuation of the Interaction Fragment prior to it, which continues to execute until completion.

Weak Sequencing The Weak Sequencing CF (WSCF) represents the behaviors of the Operands, which maintains a total order. The semantics rules of WSCF on a single Lifeline can be described as:

1. If only one EU's condition evaluates to $True$, the EU executes. The conditions of at least two EUs evaluate to $True$ make them execute sequentially.
2. The EUs continue to execute until completion.

Strict Sequencing The Strict Sequencing CF (SSCF) represents an order among the Operands that any OS in an Operand can not execute until the previous Operand completes execution. In other words, any EU may execute only if all the EUs on all the covered Lifelines within the previous Operand finish execution. SSCF enforces the synchronization among multiple Lifelines, i.e., any covered Lifeline needs to wait other Lifelines to enter the second or subsequent Operand together. The semantics rules of SSCF on a single Lifeline can be described as:

1. If only one EU's condition evaluates to $True$, the EU executes. The conditions of at least two EUs evaluate to $True$ make them execute sequentially. The second and subsequent EUs will wait until all other EUs on other Lifelines in the preceding Operand complete, and then start to execute.
2. The EUs continue to execute until completion.

Note, CRCF, ASCF, WSCF and SSCF are different from Branching, Parallel and Loop constructs which alter the graphical order of OSs, in that they do not change the order of OSs on a single Lifeline. Instead, they add more constraints to the execution of OSs. Negation CF (NCF) is the third class. The set of traces represented by the NCF is a set of invalid traces, which is the set of traces represented by its Operand.

4 Specifying a Sequence Diagram in NuSMV

Semantics rules are always the first step towards developing verification techniques and tools to analyze Sequence Diagram. In this section, we formally represent a Sequence Diagram as FSMs in terms of NuSMV **modules**. NuSMV is a model checking tool, which exhaustively explores all executions of a finite model to determine if a set of specified properties holds. For a property that does not hold, a counterexample is produced showing an error trace. A NuSMV model consists of a main module and a set of other modules, describing the FSMs. 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 at most one of the modules (including **main**) executes during a step. The initial states are defined by using **init** statements 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. 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 by using the INVAR statement, which is a boolean expression satisfied by each state.

4.1 Mapping Overview

To represent a Sequence Diagram in the NuSMV input language, we map the Lifelines into NuSMV interleaved process modules, which are instantiated and declared in the main module as the top level modules. We use a fairness constraint to enforce that each process module executes infinitely often. For each Lifeline that contains one or more CEUs, each CEU is mapped to a module inside its Lifeline's process module. These CEUs execute sequentially, adhering to their graphical order on the Lifeline. To enforce weak sequencing among CFs (i.e., the sequential execution of CEUs), we introduce a derived variable "flag_final" in each CEU to indicate if all of its OSs have executed and control can be transferred to the parent module or a following sibling module.

If a CEU is composed of multiple BEUs, they are mapped to sub-modules that are declared and instantiated inside the CEU module. Entering or leaving the EUs (i.e., the transfer of control) is determined by the Interaction Operator that composes them into the associated CF. There is an implied Weak Sequencing between EUs unless the Interaction Operator specifies otherwise (e.g., Parallel). In the case that a Sequence Diagram contains nested CFs (i.e., a CEU consists of HEUs), we map each HEU as a sub-module, whose enclosing CEUs are mapped as sub-sub-modules respectively. This procedure will be recursively applied, until all CEUs are mapped accordingly. Inter-Lifeline communication via Messages takes place when the OS of a receiving Lifeline (module) is to be executed and the corresponding OS on the sending Lifeline (specified as a condition of the receiving OS) has executed.

4.2 Basic Sequence Diagram

As described earlier, a basic Sequence Diagram is projected onto its Lifelines to obtain a BEU for each lifeline. A Message $M1$ sent from Lifeline L_i to Lifeline L_j is represented by two OSs on L_i and L_j respectively. Each NuSMV **process** module realizes one Lifeline, where the process mechanism enforces the interleaving execution of Lifelines. The module uses a boolean variable to describe an OS. Once an OS occurs, its value is set to $True$ and will stay $True$ forever. We chose this implementation over realizing an OS as an event because the process module containing the OS is interleaved with other modules. Implementing an OS as an event requires that it execute in the step immediately after the event occurs. This cannot be guaranteed since one and only one module is randomly chosen to execute in each execution step. A transition defines the execution of an OS to enforce the total order of OSs on a Lifeline and that the sending OS executes before the receiving OS associated with the same Message. In each step, at most one transition executes to change the value of OS_i to $True$ if the OSs prior to OS_i occur (set to $True$).

Figure 4 demonstrates a basic Sequence Diagram with three Lifelines. Figure 7 shows its NuSMV description, which contains a main module for the Sequence Diagram. The three Lifelines, are declared as instances of modules, *l_L1*, *l_L2*, and *l_L3*. Space limitations allow us to only show the implementation of Module *L1*. The VAR section declares two OSs, *OS_sx* and *OS_rx*, as two boolean variables, where r and s denote they are sending or receiving OSs, and x is the corresponding Message number. The ASSIGN section defines the transition relation using next statements. For example, the receiving OS, *OS_r3*, will execute (set to $TRUE$ in NuSMV model) if the sending OS of Message 3 and the preceding OS, *OS_s1*, on Lifeline $L1$ have executed.

```
MODULE main
  VAR
    l_L1: process L1;
    l_L2: process L2;
    l_L3: process L3;
MODULE L1 (L2, L3)
  DEFINE
    flag_final := OS_r3;
  VAR
    OS_s1:boolean;
    OS_r3:boolean;
```

```
ASSIGN
  init(OS_s1) := FALSE;
  next(OS_s1) := case
    !OS_s1                :TRUE;
    1                     :OS_s1;
   esac;
  init(OS_r3) := FALSE;
  next(OS_r3) := case
    OS_s1 & L3.OS_s3      :TRUE;
    1                     :OS_r3;
   esac;
```

Fig. 7. Basic Sequence Diagram to NuSMV

4.3 Translating Combined Fragments

A CF enclosing multiple Lifelines is projected onto each Lifeline as a CEU that contains BEUs for all the Operands respectively. The CEU on each Lifeline is represented as a NuSMV submodule of the Lifeline module, with BEUs as the sub-submodules until all CEUs and EUs are represented accordingly. The composition of the child modules is implemented with an extra variable indicating the execution status of each child module, such as “start”, “complete”, or “selected”. The extra variable facilitates control transfer among multiple modules based on the meanings of each Interaction Operator. The structure of the Sequence Diagram is preserved during translation.

Concurrency PCF can be captured by NuSMV’s interleaving process modules directly . We only show the module of the CEU on Lifeline *L1* of Figure 1 in Figure 8. Two module instances corresponding to the two Operands’ BEUs are created as NuSMV executing and interleaving processes. The BEUs can be translated to modules as a simple Sequence Diagram. We show the first BEU of *L1* as a NuSMV module in figure 13.

```
MODULE par_L1 (par_L2, par_L3)
 VAR
   op_par_op1_L1:process par_op1_L1(flag_par, par_L2.op_par_op1_L2, par_L3.op_par_op1.L3);
   op_par_op2_L1:process par_op2_L1(flag_par, par_L2.op_par_op2_L2, par_L3.op_par_op2.L3);
```

Fig. 8. NuSMV module for PCF

Branching The ALCF is translated into CEU submodules, one for each Lifeline, containing the EU sub-submodules, one for each Operand. The Interaction Constraint is translated to a condition for each sub-submodule. An enumerated variable “flag_alt” is used to capture the choice of the sub-submodule to execute on each Lifeline. A boolean variable “exe” is introduced for each Operand, and is set to $True$ if the Operand is chosen to execute. The constraint under INVAR restricts that the Operand’s “exe” can be set to $True$ only if the Operand’s *cond* evaluates to $True$, and at most one “exe” can be set to $True$. The use of “exe” guarantees that all the enclosed Lifelines choose the same Operand’s sub-submodules to execute to avoid inconsistent choices (e.g., Lifeline L1 chooses Operand 1 whereas Lifeline L2 chooses Operand 2).

Figure 9 is an example of an ALCF with three Operands enclosing three Lifelines. Figure 10 shows the NuSMV submodule of the ALCF’s CEU on Lifeline *L2*. Three sub-submodules are instantiated to represent three BEUs respectively. “flag_alt” is initialized to -1, and then is set depending on the values of the condition in each BEU. If none of the Operands are chosen (i.e., none of the conditions evaluate to $True$) then the flag is set to “3” and control can be transferred to the following OS or CEU. The flag is set to one of the values $\{0, 1, 2\}$ depending on the value of “exe” for each Operand, expressing that if the three Interaction Operands have not been executed and all three Interaction Constraints evaluate to $True$, then any Operand may be chosen to execute.

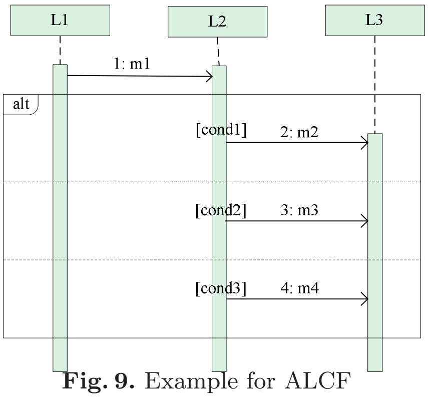

Fig. 9. Example for ALCF

```
MODULE alt_L2 (OS_r1, alt_L1, alt_L3, cond1, cond2, cond3)
 VAR
  flag_alt: -1..3;
  op1_L2:alt_op1_L2(L3.op1_L3,flag_alt);
  op2_L2:alt_op2_L2(L3.op2_L3,flag_alt);
  op3_L2:alt_op3_L2(L3.op3_L3,flag_alt);
 ASSIGN
  ...
  init(flag_alt) := -1;
  next(flag_alt) := case
   flag_alt=-1 & OS_r1 & op1_L2.exe           : 0;
   flag_alt=-1 & OS_r1 & op2_L2.exe           : 1;
   flag_alt=-1 & OS_r1 & op3_L2.exe           : 2;
   flag_alt=-1 & OS_r1 & !cond1 & !cond2 & !cond3 :3;
    (flag_alt=0 & op1_L2.flag_final)
   |(flag_alt=1 & op2_L2.flag_final)
   |(flag_alt=2 & op3_L2.flag_final)          : 3;
   1                                          :flag_alt;
   esac;
  ...
INVAR
 ((l_L2.alt.op1_L2.exe->cond1)&(l_L2.alt.op2_L2.exe->cond2)&(l_L2.alt.op3_L2.exe->cond3))
 & ((l_L2.alt.op1_L2.exe & !l_L2.alt.op2_L2.exe & !l_L2.alt.op3_L2.exe)|(!l_L2.alt.op1_L2.exe
 & l_L2.alt.op2_L2.exe & !l_L2.alt.op3_L2.exe)|(!l_L2.alt.op1_L2.exe & !l_L2.alt.op2_L2.exe
 & l_L2.alt.op3_L2.exe)|(!l_L2.alt.op1_L2.exe & !l_L2.alt.op2_L2.exe & !l_L2.alt.op3_L2.exe))
```

Fig. 10. NuSMV module for ALCF

The OCF is translated into NuSMV module with similar strategy as ALCF. An enumerated variable “flag_opt” is used to describe the execution of the EU sub-sub-module. We show an example of OCF in figure 11. Figure 12 represents the NuSMV submodule of the OCF’s CEU and the sub-sub-module of the EU on Lifeline $L2$. “flag_opt” is initialized to -1, and then is set depending on the values of the condition. If the condition evaluates to $True$, “flag_opt” is set to 0 to make the sub-sub-module can be executed. Otherwise, ‘flag_opt” is set to 1 to skip the sub-sub-module.

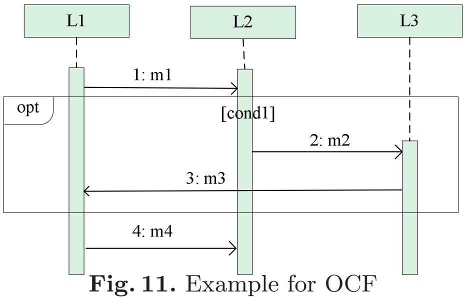

Fig. 11. Example for OCF

```
MODULE opt_L2(OS_r1, cond1)
 VAR
  flag_opt : -1..1;
  op1_L2 : opt_op1_L2(flag_opt);
 ASSIGN
  init(flag_opt) := -1;
  next(flag_opt) := case
   flag_opt=-1 & OS_r1 & cond1  :0;
   flag_opt=-1 & OS_r1 & !cond1 :1;
   flag_opt=-1 & OS_r1 & cond1
   & op1_L2.flag_final          :1;
  esac;
MODULE opt_op1_L2(flag_opt)
 DEFINE
  flag_final := OS_s2;
 VAR
  OS_s2:boolean;
 ASSIGN
  init(OS_s2) := FALSE;
  next(OS_s2) := case
    flag_opt=0   :TRUE;
    1            :OS_s2;
   esac;
  ...
```

Fig. 12. NuSMV module for OCF

The BCF has been rewritten to ALCF with two Operands as we described in section 3.3. Therefore, the BCF can be translated to NuSMV module as ALCF.

Atomicity In the NuSMV module for the CRCF, a boolean variable “isCritical” is introduced to restrict the OSs within the CRCF on each Lifeline from being interleaved with other OSs in other modules on the same Lifeline. Figure 13 shows the NuSMV representation for an HEU of a PCF containing a CRCF (see figure 1). The variable “isCritical” is initialized to $False$ and is set to $True$ when the CEU of CRCF on Lifeline $L3$ is entered. It remains $True$ until the CEU finishes execution. Once the CEU completes, “isCritical” is set to $False$, and the other OSs on Lifeline $L3$ may execute. The negation of “isCritical” is used as a constraint for the execution of each OS in the other modules except the CEU.

```
MODULE par_op1_L3(flag_par, par_op1_L2, cond1, cond3)
 DEFINE
  flag_final := OS_r4;
 VAR
  OS_s3:boolean;
  OS_r4:boolean;
  isCritical:boolean;
 ASSIGN
  init(OS_s3) := FALSE;
  next(OS_s3) := case
    cond1 & flag_par=0
  & par_op1_L2.OS_s2 & cond3   :TRUE;
    1                          :OS_s3;
   esac;
```

```
  init(OS_r4) := FALSE;
  next(OS_r4) := case
    OS_s3 & par_op1_L2.OS_s4    :TRUE;
    1                           :OS_r4;
   esac;
  init(isCritical) := FALSE;
  next(isCritical) := case
    OS_s3 & !OS_r4              :TRUE;
    OS_r4                       :FALSE;
    1                           :isCritical;
   esac;
```

Fig. 13. NuSMV module for CRCF

Iteration We represent the fixed, bounded LCF with a NuSMV model, where the loop body iterations are composed with the Weak Sequencing Operator. The NuSMV module in figure 15 is a loop module of the lifeline L_1 in figure 14. *Count* is the variable to calculate iteration times, which is zero initially. When all the Lifelines finish current iteration, *count* increase one as the start of the next iteration. At the start of each iteration, the loop condition and *count* are checked, and the flags of each OS set to *False*. If the condition is evaluated to *False* or *count* is over *maxint*, new iteration can not start, loop modules for Lifelines reach their final states and the execution of loop combined fragment is over.

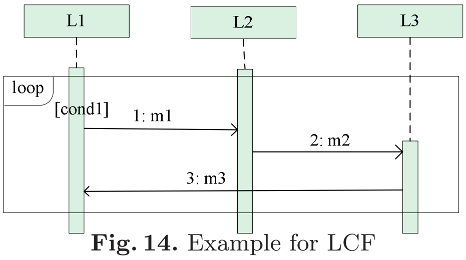

Fig. 14. Example for LCF

```
MODULE L1(L2, L3, cond, min, max)
 DEFINE
  flag_final := flag_r3 & (count=3);
 VAR
  flag_s1 : boolean;
  flag_r3 : boolean;
  count : 0..3;

 ASSIGN
  init(flag_s1) := FALSE;
  next(flag_s1) := case
    (count>0) & !flag_s1           :TRUE;
    (next(count) = count+1) & flag_s1 : FALSE;
    1          :flag_s1;
   esac;
  init(flag_r3) := FALSE;
  next(flag_r3) := case
    (count>0) & !flag_r3 & L3.flag_s3 & flag_s1 :TRUE;
    (next(count) = count+1) & flag_r3 : FALSE;
    1          :flag_r3;
   esac;
  init(count)    := 0;
  next(count)    := case
    !flag_s1 & (count < min) & (count < 3) : count+1;
    flag_r3 & (count < min) & (count < 3) : count+1;
    flag_r3 & (count >= min) & (count < max) & cond & (count < 3) : count+1;
    flag_r3 & (count >= min) & (count < max) & !cond & (count < 3) : count;
    flag_r3 & (count >= max) : count;
    1           : count;
   esac;
    ...
```

Fig. 15. NuSMV module for LCF

Assertion ASCF has a single Operand. The translation strategy is very similar to the translation of CRCF. Projecting the ASCF onto its involved Lifelines obtains a collection of CEUs, one for each Lifeline. In the NuSMV modules for an ASCF, a boolean variable "inAssertion" is introduced for each Lifeline module to indicate the start of the ASCF CEU. This restricts the OSs within the CEU from being interleaved with OSs in other regions once the CEU starts to execute. The variable is *False* initially and is set to *True* when the OSs in the set of pre(CEU) executed. The variable is set to "False" once the CEU completes. Figure 17 shows the NuSMV module of the CEU on Lifeline $L2$ in figure 16.

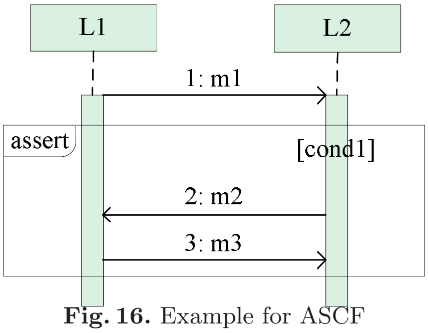

Fig. 16. Example for ASCF

```
MODULE L2(L1, cond1)
 DEFINE
  flag_final := OS_r3;
 VAR
  OS_r1:boolean;
  OS_s2:boolean;
  OS_r3:boolean;
  inAssertion:boolean;
 ASSIGN
  init(OS_r1) := FALSE;
  next(OS_r1) := case
    L1.OS_s1 & !inAssertion :TRUE;
    1                       :OS_r1;
   esac;
  init(OS_s2) := FALSE;
  next(OS_s2) := case
    cond1 & OS_r1           :TRUE;
    1                       :OS_s2;
   esac;
  init(OS_r3) := FALSE;
  next(OS_r3) := case
    OS_s2 & L1.OS_s3        :TRUE;
    1                       :OS_r3;
   esac;
  init(inAssertion) := FALSE;
  next(inAssertion) := case
    cond1 & OS_r1           :TRUE;
    OS_r3                   :FALSE;
    1                       :inAssertion;
   esac;
  ...
```

Fig. 17. NuSMV module for ASCF

Weak Sequencing and Strict Sequencing Translating the WSCF into NuSMV module describes the total order among EUs. The variable "flag_final"of each EU's sub-sub-module is considered as a condition for another EU's sub-sub-module which executes right after it on the same Lifeline. Figure 18 is an example of a WSCF. Figure 19 shows a NuSMV sub-sub-module, which is translated for the EU of WSCF's second Operand on Lifeline $L2$. It takes the "flag_final" of the first Operand's EU as condition, which enforces the order between two sub-sub-modules.

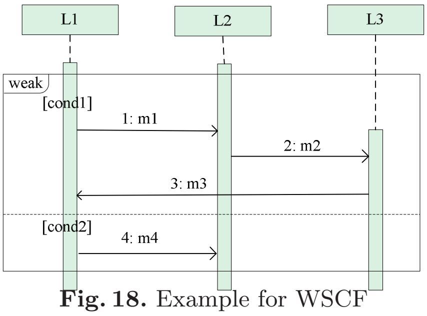

Fig. 18. Example for WSCF

```
MODULE weak_op2_L2(weak_op2_L1,
                   weak_op1_L2, cond2)
 DEFINE
  flag_final := OS_r4;
 VAR
  OS_r4:boolean;
 ASSIGN
  init(OS_r4) := FALSE;
  next(OS_r4) := case
    cond2 & weak_op2_L1.OS_s4
    & weak_op1_L2.flag_final    :TRUE;
    1                           :OS_r4;
   esac;
  ...
```

Fig. 19. NuSMV module for WSCF

In the NuSMV module of Strict SSCF, an EU's sub-sub-module (not in the first Operand) needs the "flag_final"of every EU's sub-sub-module within the previous Operand to be its condition. This assert that the EU can not execute until the previous Operand completes execution. Figure 20 is an example of a SSCF and figure 21 shows the NuSMV sub-sub-module for the EU of SSCF's second Operand on Lifeline $L2$. The sub-sub-module can execute if each EU's "flag_final" within the first Operand sets to $True$, which enforce the order among Operands.

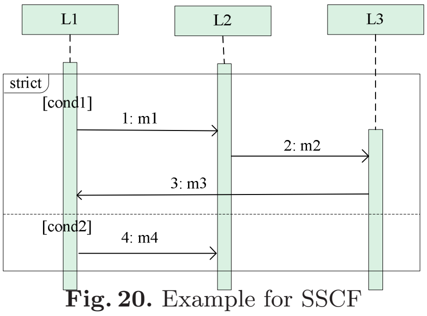

Fig. 20. Example for SSCF

```
MODULE strict_op2_L2(strict_op2_L1, cond2,
strict_op1_L1, strict_op1_L2, strict_op1_L3)
 DEFINE
  flag_final := OS_r4;
 VAR
  OS_r4:boolean;
 ASSIGN
  init(OS_r4) := FALSE;
  next(OS_r4) := case
    cond2 & strict_op2_L1.OS_s4
    & strict_op1_L1.flag_final
    & strict_op1_L2.flag_final
    & strict_op1_L3.flag_final :TRUE;
    1                       :OS_r4;
   esac;
  ...
```

Fig. 21. NuSMV module for SSCF

Coregion The Coregion is an area containing OSs or other nested CEUs which can be interleaved in any order on a single Lifeline. We represent the Coregion as a PCF in NuSMV input language. Each OS or nested CEU in the Coregion is considered as an Operand of the PCF.

General Ordering A General Ordering represents the order between two unordered OSs, which describes one OS must happen before the other OS. In NuSMV input language, we show the General Ordering as the structure of a Message, which maps the prior OS as a condition of the latter OS.

Interaction Use Interaction Use embeds the content of the referred Interaction into the specified Interaction, thus composing a single, larger Interaction. For each Lifeline that contains one or more Interaction Use, each CEU of the referred Interaction is mapped to a sub-module inside the CEU module of the specified Interaction. These sub-modules execute sequentially, adhering to their graphical order on the Lifeline.

5 Specifying a Sequence Diagram in LTL

The previous section has demonstrated translating Sequence Diagrams into NuSMV models. Representing a Sequence Diagram as NuSMV modules allows the generation of all possible traces of the Diagram. In this section, we describe how to use LTL formulas to represent the constraints on event traces implied by Sequence Diagrams. We have verified that the NuSMV modules satisfy the LTL representation of the Sequence Diagrams. This suggests that our two types of formal representations of a Sequence Diagram conform to each other.

LTL is a formal language for expressing properties related to a sequence of states in terms of temporal logic operators (e.g., $\bigcirc$, $\Diamond$, and $\Box$) and logical connectives (e.g., $\wedge$ and $\vee$). $\Box p$ 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. $p\ \mathcal{U}\ q$ 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 $True$.

We use an LTL formula to express the partial orders of events (OSs) imposed by different constructs. It can be a challenge to consider all semantic constraints at once as they may interfere with each other. To conquer this complexity, we devised a novel template, comprised of simpler definitions to represent each semantic aspect as a separate concern. We show that the simpler definitions can be composed using logical conjunctions to represent a complex Sequence Diagram.

5.1 Auxiliary Function

To facilitate the translation of a Sequence Diagram into an LTL specification, we define a collection of auxiliary functions in Table 1. 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 Lifeline.

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 5. 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 Interaction Fragment v prior to u on the same Lifeline:

Function	Explantation
$MSG(p)$	return the set of all Messages directly enclosed in p
$MSGOS(e)$	return the associate Message of OS e
$LN(p)$	return the set of all Lifelines in p
$AOS(q)$	return the set of all OSs in q
$TAOS(q)$	return a set of OSs which are enabled and chosen to execute in q
$SND(j)$	return the sending OS of Message j
$RCV(j)$	return the receiving OS of Message j
$CND(op)$	return a boolean value representing the Interaction Constraint of Interaction Operand *op*, which is lifted to a Combined Fragment containing an sole Operand
$OPND(u)$	return the set of all Interaction Operands in Combined Fragment u
$TOP(u)$	return a set of Interaction Operands whose Constraints evaluate to $True$ within Combined Fragment u, i.e. $\{op \vert op \in OPND(u) \wedge CND(op) = True\}$
$ABEU(u)$	return a set of BEUs directly contained by CEU or EU u
$TBEU(u)$	return a set of BEUs whose Constrains evaluate to $True$ and which are directly contained by CEU or EU u, i.e. $\{beu \vert beu \in ABEU(u) \wedge CND(beu) = True\}$
$TOS(u)$	return a set of OSs of the BEUs directly enclosed in CEU u whose Constaints evaluates to $True$, i.e. $\{os \vert beu \in TBEU(u) \wedge os \in AOS(beu)\}$
$pre(u)$	return a set of OSs which may happen right before CEU u
$post(u)$	return a set of OSs which may happen right after CEU u
$nested(u)$	return a set of Combined Fragments, 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.
$typeOS(m, l)$	return the type of OS for Message m on Lifeline l , which is a sending OS or a receiving OS.
$OS(m, l)$	return the associate OS of Message m on Lifeline l.
$typeCF(u)$	return the type of CF u.

Table 1. Auxiliary functions

- Case1: If v is a BEU, *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;
  - 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*, we check the Interaction Fragment 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, we can recursively apply cases 1 to 5 to v and its nested CEUs. *post(u)* can be calculated in a similar way.

5.2 Basic Sequence Diagram

Recall Two semantic aspects must be considered for the basic Sequence Diagram.

1. On each Lifeline, OSs execute in their graphical order.

2. For the same Message, sending must take place before receiving.

Working towards similar goals, Kugler et al. [10] and Kumar et al. [12] have presented translations from a Live Sequence Chart (LSC) to LTL formulas. We adapt their previous work and devise an LTL template $\mathit{\Phi}_{seq}^{Basic}$ for a basic Sequence Diagram seq (see figure 22). The template is a conjunction of the formulas α_i and β_i to define the above two rules respectively. α_i describes the total order of OSs on a single Lifeline i , i.e., for all $k \geq 1$, OS_k must happen (strictly) before OS_{k+1}, ensured by $\neg p \, \widetilde{\mathcal{U}} \, q$. Also, the last OS, $OS_{|AOS(i)|}$ must happen, in which $|s|$ has its usual meaning, denoting the size of set s. β_j defines the receiving OS, $RCV(j)$, cannot happen until the sending OS of Message j, $SND(j)$ happens.

$$\begin{aligned}
\mathit{\Phi}_{seq}^{Basic} &= \bigwedge_{i \in LN(seq)} \alpha_i \wedge \bigwedge_{j \in MSG(seq)} \beta_j \\
\alpha_i &= (((|AOS(i) > 1) \wedge (\bigwedge_{k \in [1..|AOS(i)|-1]} (\neg OS_{k+1} \, \widetilde{\mathcal{U}} \, OS_k))) \vee (|AOS(i) = 1)) \wedge \Diamond OS_{|AOS(i)|} \\
\beta_j &= \neg RCV(j) \, \widetilde{\mathcal{U}} \, SND(j)
\end{aligned}$$

Fig. 22. Basic Sequence Diagram to LTL

5.3 Translating Combined Fragments

An LTL template $\mathit{\Phi}_{CF}$, represents the semantics of (nested) Combined Fragment CF. $\mathit{\Phi}_{CF}$ (see figure 23) considers two cases. Formula (1) asserts the case that CF contains no nested CFs. If the CF contains at least one Operand whose condition evaluates to $True$, then we use formula $\mathit{\Psi}^{CF}$, to define the semantics of CF, otherwise, sub-formula η^{CF} enforces Weak Sequencing (i.e. *CF's* preceding Interaction Fragments must take place before the succeeding ones). Formula (2) asserts the case that CF contains nested CFs. The only difference between formulas (1) and (2) lies in that (2) has to define each CF_i, which are directly enclosed in CF, using template $\mathit{\Phi}_{CF_i}$. In this way, $\mathit{\Phi}_{CF}$ can be defined recursively until it has no nested CFs.

$$\begin{aligned}
\mathit{\Phi}^{CF} &= \begin{cases} ((\bigvee_{n \in OPND(CF)} CND(n)) \wedge \mathit{\Psi}^{CF}) \vee (\neg(\bigvee_{n \in OPND(CF)} CND(n)) \wedge \eta^{CF}) & (1) \\ ((\bigvee_{n \in OPND(CF)} CND(n)) \wedge \mathit{\Psi}^{CF} \wedge \bigwedge_{\substack{CF_i \in nested(CF) \\ typeCF(CF) \neq ALCF}} \mathit{\Phi}^{CF_i}) \vee (\neg(\bigvee_{n \in OPND(CF)} CND(n)) \wedge \eta^{CF}) & (2) \end{cases} \\
\eta^{CF} &= \bigwedge_{i \in LN(CF)} ((\bigwedge_{OS \in post(CF\uparrow_i)} (\neg OS)) \widetilde{\mathcal{U}} (\bigwedge_{OS_p \in pre(CF\uparrow_i)} OS_p)) \\
\theta_k &= \bigwedge_{i \in LN(k)} (\bigwedge_{g \in TBEU(k\uparrow_i)} \alpha_g) \wedge \bigwedge_{j \in MSG(TOP(k))} \beta_j \\
\alpha_g &= (((|AOS(g) > 1) \wedge (\bigwedge_{k \in [r+1..r+|AOS(g)|-1]} (\neg OS_{k+1} \, \widetilde{\mathcal{U}} \, OS_k))) \vee (|AOS(g) = 1)) \wedge \Diamond OS_{r+|AOS(g)|} \\
\gamma_i^{CF} &= \bigwedge_{OS \in TOS(CF\uparrow_i)} ((\neg OS \, \widetilde{\mathcal{U}} \, (\bigwedge_{OS_{pre} \in pre(CF\uparrow_i)} OS_{pre})) \wedge ((\bigwedge_{OS_{post} \in post(CF\uparrow_i)} (\neg OS_{post})) \, \widetilde{\mathcal{U}} \, OS))
\end{aligned}$$

Fig. 23. LTL templates for Combined Fragment

Recall CF modifies the execution order of OSs on each covered Lifeline by the following semantic rules.

1. Interaction Fragments, including OSs and CFs, are combined using Weak Sequencing.
2. The semantics of each CF Operator (summarized in Section 2) defines the execution of all the Operands. Each InteractionOperator has its specific semantic implications on the execution of events on the covered Lifeline.
3. Within a CF, the order of Interaction Fragments within each Operand are maintained if the constraint of the Operand evaluates to $True$, otherwise the CF is excluded.

Formula Ψ^{CF} captures the semantics of each individual Combined Fragment CF using a conjunction of sub-formula γ_i^{CF}, enforcing semantic rule 1, and sub-formula θ_k, enforcing semantic rule 3. γ_i^{CF} enforces the sequential execution on every Lifeline i. 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 (see appendix for details). 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. Conjunct 1 or conjunct 2 sets to $True$ if either $pre(CF\uparrow_i)$ or $post(CF\uparrow_i)$ returning empty set. θ_k defines the order among OSs within CF or Interaction Operand k, which is a conjunction of α_g and β_j, where g is a BEU on Lifeline i, whose condition evaluates to $True$. The definition of α_g is similar to α_i, which enforces the total order of a finite trace of OSs $\sigma_u[(r+1)..(r+|AOS(g)|)]$ of length $|AOS(g)|$ within BEU g. The semantic specifics for different types of CF Operators are defined as below.

Concurrency θ_{PCF} and γ_i^{PCF} (defined in the previous paragraph) in Ψ^{PCF} precisely describe the semantics of PCF, where Operands may be interleaved (see figure 24).

$$\Psi^{PCF} = \theta_{PCF} \wedge \bigwedge_{i \in LN(PCF)} \gamma_i^{PCF}$$

Fig. 24. LTL formula for PCF

Branching The translation of an ALCF into an LTL formula must enumerate all possible choices of executions in that only OSs of one of the Operands, whose conditions evaluate to $True$, will happen. Figure 25 is the LTL formula with two conjunctive sub-formulas. We define the logical operator "unique or" as "$\widehat{\bigvee}$", to denote that exactly one of its Operands is chosen. To indicate the chosen Operand, we use a boolean variable "exe" for each Operand. The first conjunct expresses that exactly one Operand is chosen from $TOP(ALCF)$ and only "exe" of the chosen Operand sets to $True$ if at least one of the Interaction Constraints evaluates to $True$. m is one of Operands whose conditions evaluate to $True$ within $ALCF$. In the second conjunct, $\bar{\theta}_m$ defines the chosen Operand, which includes the partial order of OSs and the semantics of CFs directly enclosed in the chosen Operand. Functions $\bar{\theta}_m$ and Φ^{CF} invoke each other to form indirect recursion. The Weak Sequencing of the chosen Operand is represented by $\bar{\gamma}_{i,m}^{ALCF}$ instead of γ_i^{CF}, which enforces Weak Sequencing of the chosen Operand.

$$\begin{aligned}
\Psi^{ALCF} =& (\widehat{\bigvee_{m \in TOP(ALCF)}} exe_m) \wedge \bigwedge_{m \in TOP(ALCF)} (exe_m \rightarrow (\bar{\theta}_m \wedge \bigwedge_{i \in LN(ALCF)} \bar{\gamma}_{i,m}^{ALCF})) \\
\bar{\theta}_m =& \bigwedge_{i \in LN(m)} (\bigwedge_{g \in TBEU(m\uparrow_i)} \alpha_g) \wedge \bigwedge_{j \in MSG(TOP(m))} \beta_j \wedge \bigwedge_{CF_i \in nested(m)} \Phi^{CF_i} \\
\bar{\gamma}_{i,m}^{ALCF} =& \bigwedge_{OS \in TOS(m\uparrow_i)} ((\neg OS\, \widetilde{\mathcal{U}} (\bigwedge_{OS_{pre} \in pre(ALCF\uparrow_i)} OS_{pre})) \wedge ((\bigwedge_{OS_{post} \in post(ALCF\uparrow_i)} (\neg OS_{post}))\, \widetilde{\mathcal{U}}\, OS))
\end{aligned}$$

Fig. 25. LTL formula for ALCF

Figure 26 represents the LTL interpretation of an OCF. θ_{OCF} and γ_i^{OCF} in Ψ^{OCF} precisely describe the order among OSs within the OCF and the Weak Sequencing between the OCF and its preceding/succeeding Interaction Fragment.

$$\Psi^{OCF} = \theta_{OCF} \wedge \bigwedge_{i \in LN(OCF)} \gamma_i^{OCF}$$

Fig. 26. LTL formula for OCF

We have rewritten the BCF as an ALCF in section 3.3, thus, the LTL representation of BCF applies the LTL formula for ALCF with two Operands.

Atomicity Formula $\Psi^{CRCF}_{critical}$ presents the semantics for CRCF (see figure 27). θ_{CRCF} and γ_i^{CRCF} have their usual meanings. δ_{M_1,M_2} enforces that on a single Lifeline, if any OS in set M_1 occurs, no OSs in set 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 Combined Fragment $CRCF$ from the set of OSs for Sequence Diagram seq on Lifeline i.

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

$$\Psi^{CRCF} = \bigwedge_{i \in LN(CRCF)} \delta_{(TAOS(CRCF\uparrow_i),(TAOS(seq\uparrow_i)\backslash TAOS(CRCF\uparrow_i)))} \wedge \theta_{CRCF} \wedge \bigwedge_{i \in LN(CRCF)} \gamma_i^{CRCF}$$

Fig. 27. LTL formula for CRCF

Iteration To represent LCF, we introduce R, representing the number of iterations and $count_a$, representing the current iteration number on Lifeline i. Fixed LCF can be unfolded by repeating iterations, which is represented as $LCF[count_a]$ for LCF in iteration $count_a$. Each OS has an array and the value of $count_i$ represents each element in the array. An OS and the value of $count_i$ together represent the OS in a specific iteration, (e.g., the element $(OS_k[count_3])$ expresses OS_k in the third iteration). Figure 28 show an LTL for a LCF. $\widehat{\theta}$ overload θ_k, which asserts the order of OSs during each iteration. $\widehat{\gamma}_i$ enforces the Weak Sequencing among LCF and its preceding/following sets of OS on each Lifeline i. Template κ_i is introduced to enforce Weak Sequencing among iterations.

Negation A Negative Combined Fragment (NCF) represents that the set of traces within an NCF are invalid. (Recall we constrain that NCF should be at the top level of a Sequence Diagram.) Formula $\Psi^{NCF} = \theta_{NCF}$ enforces the order of OSs in the BEUs directly enclosed in NCF with θ_{NCF}. If the Interaction Constraint of the NCF evaluates to $False$, the traces within the NCF are not invalid traces. Note, these traces may not be valid traces either.

Assertion Formula Ψ^{ASCF} defines Assertion Combined Fragment (ASCF), representing a set of mandatory traces, which are the only valid traces following OSs in $pre(ASCF \uparrow_i)$ on Lifeline i, formally $\Psi^{ASCF} = \theta_{ASCF} \wedge \bigwedge_{i \in LN(ASCF)} \gamma_i^{ASCF}$

Weak Sequencing The LTL formula representation of a WSCF is shown in figure 29. θ_{WSCF} and γ_i^{WSCF} have their usual meanings. Template $\mu_{k,i}$ is introduced to enforce the total order among the EU of Operand k and its preceding/succeeding EU on Lifeline i when the Constraint of Operand k

$$
\begin{aligned}
\Psi^{LCF}_{loop} =& \widehat{\theta} \wedge \bigwedge_{i \in LN(LCF)} \widehat{\gamma}_i \wedge \bigwedge_{i \in LN(LCF)} \kappa_i \\
\widehat{\theta} =& \bigwedge_{i \in LN(LCF)} \Big(\bigwedge_{g \in TBEU(LCF\uparrow_i)} \bar{\alpha}_g \Big) \wedge \bigwedge_{j \in MSG(TOP(k))} \bar{\beta}_j \\
\bar{\alpha}_i =& \bigwedge_{\substack{k \in [r+1..r+|AOS(g)|-1] \\ count_i \in [1..R]}} ((\neg(OS_{k+1}[count_i]))\widetilde{\mathcal{U}}(OS_k[count_i])) \wedge \bigwedge_{count_i \in [1..R]} (\Diamond(OS_{r+|AOS(g)|}[count_i])) \\
\bar{\beta}_j =& \bigwedge_{count_i \in [1..R]} (\neg RCV(j)[count_i])\widetilde{\mathcal{U}}(SND(j)[count_i]) \\
\widehat{\gamma}_i =& \bigwedge_{OS \in TOS(LCF[1]\uparrow_i)} (\neg OS\ \widetilde{\mathcal{U}}\ (\bigwedge_{OS_{pre} \in pre(LCF[1]\uparrow_i)} OS_{pre})) \\
&\wedge \bigwedge_{OS \in TOS(LCF[R]\uparrow_i)} ((\bigwedge_{OS_{post} \in post(LCF[R]\uparrow_i)} (\neg OS_{post}))\ \widetilde{\mathcal{U}}\ OS) \\
\kappa_i =& \bigwedge_{count_i \in [1..R-1]} (\bigwedge_{OS_q \in TAOS(LCF\uparrow_i)} (\neg(OS_q[count_i+1]))\widetilde{\mathcal{U}}(\bigwedge_{OS_p \in TAOS(LCF\uparrow_i)} OS_p[count_i])
\end{aligned}
$$

Fig. 28. LTL formula for fixed LCF

evaluates to $True$. Functions $pre(u)$ and $post(u)$ are overloaded to an EU u, which returns the set of OSs happening right before or right after EU u. If the Constraint of Operand k evaluates to $False$, the order between EU u's preceding and succeeding EUs are established, that the preceding EU must take place before the succeeding one.

$$
\begin{aligned}
\Psi^{WSCF}_{weak} =& \theta_{WSCF} \wedge \bigwedge_{i \in LN(WSCF)} \gamma_i^{WSCF} \wedge \bigwedge_{i \in LN(WSCF)} (\bigwedge_{k \in OPND(WSCF)} \mu_{k,i}) \\
\mu_{k,i} =& (CND(k) \wedge (\bigwedge_{OS \in TOS(k\uparrow_i)} ((\neg OS\ \widetilde{\mathcal{U}}\ (\bigwedge_{OS_{pre} \in pre(k\uparrow_i)} OS_{pre})) \wedge ((\bigwedge_{OS_{post} \in post(k\uparrow_i)} (\neg OS_{post}))\ \widetilde{\mathcal{U}}\ OS)))) \\
&\vee(\neg CND(k) \wedge ((\bigwedge_{OS_{post} \in post(k\uparrow_i)} (\neg OS_{post}))\widetilde{\mathcal{U}}(\bigwedge_{OS_{pre} \in pre(k\uparrow_i)} OS_{pre})))
\end{aligned}
$$

Fig. 29. LTL formula for WSCF

Strict Sequencing Figure 30 represents the LTL formula of an SSCF. Function χ_k is introduced to assert the total order Operand k and its preceding/succeeding Operands when the Constraint of Operand k evaluates to $True$. Otherwise, Operand k's preceding and succeeding Operands are ordered, i.e., the succeeding Operand can not happen until the preceding one completes.

Coregion We translate the Coregion into LTL formula in a similar way as a PCF. Each OS in the Coregion is considered as an Operand of the PCF. Figure 31 show an LTL for a Coregion on Lifeline i. $\gamma_i^{coregion}$ describes the Weak Sequencing between coregion and its preceding/succeeding set of OSs. Function $MSGOS(OS_k)$ return the associate Message of OS_k and template β enforces the order of OSs within the Message.

General Ordering In LTL formula, we specify the two OSs of General Ordering as a pair of ordered OSs. Figure 32 show an LTL for a Sequence Diagram with a General Ordering(GO). OS_p and OS_q

$$\begin{aligned}\Psi^{SSCF}_{strict} &= \theta_{SSCF} \wedge \bigwedge_{i \in LN(SSCF)} \gamma_i^{SSCF} \wedge \bigwedge_{k \in OPND(SSCF)} \chi_k \\ \chi_k =&(CND(k) \wedge ((\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS \in TOS(k\uparrow_i)} (\neg OS))) \,\widetilde{\mathcal{U}}\, (\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS_{pre} \in pre(k\uparrow_i)} OS_{pre}))) \\ &\wedge ((\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS_{post} \in post(k\uparrow_i)} (\neg OS_{post}))) \,\widetilde{\mathcal{U}}\, (\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS \in TOS(k\uparrow_i)} OS)))) \\ &\vee(\neg CND(k) \wedge ((\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS_{post} \in post(k\uparrow_i)} (\neg OS_{post})))\widetilde{\mathcal{U}}(\bigwedge_{i \in LN(SSCF)} (\bigwedge_{OS_{pre} \in pre(k\uparrow_i)} OS_{pre}))))\end{aligned}$$

Fig. 30. LTL formula for SSCF

$$\Psi_i^{coregion} = \gamma_i^{coregion} \wedge \bigwedge_{OS_k \in TOS(coregion)} \beta_{MSGOS(OS_k)}$$

Fig. 31. LTL formula for Coregion

are two OSs connected by the General Ordering, which specifies that OS_q can not executes until OS_p executes. We don't consider the case when General Ordering crossing the border of Interaction Fragments.

$$\Psi^{GO} = \neg OS_q \; \mathcal{U} \; OS_p$$

Fig. 32. LTL formula for General Ordering

Discussion Translating a Sequence Diagram *seq* into LTL formulas has two cases (See figure 33): (1) If *seq* contains a CF, the order of OSs within BEUs directly contained in *seq* are represented by α and β. Φ_{CF} represents each CF at the top level, appended as a conjunct. (2) If *seq* is a basic Sequence Diagram, the LTL formula is shown as Φ_{seq}^{Basic}. This paper focuses on defining the semantics of interactions in terms of event traces. We assume the Interaction Constraints of CFs are evaluated at the beginning of the execution of a Sequence Diagram since they are not modified by simple events without parameters.

Interaction Use Figure 34 show an LTL for a Sequence Diagram with an Interaction Use. In Ψ_{ref}, the first conjunct describes that the referred Sequence Diagram obey the order defined by formula $\Phi_{seq_{ref}}$ (see figure 33 for its definition). The second conjunct enforces that the referred Sequence Diagram and its adjacent OSs are ordered by Weak Sequencing, which is represented by $\gamma_i^{seq_{ref}}$.

6 Verifying Safety and Consistency Properties

Formalizing Sequence Diagrams in both NuSMV modules and LTL specifications permits us to leverage the analytical powers of model checking to justify the conformance of the two translation processes. Now we present how to model check one Sequence Diagram, represented as a NuSMV model, with respect to another Sequence Diagram, represented as an LTL property.

6.1 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 that none of the forbidden traces within an NCF exist as sub-traces within

$$\Phi_{seq} = \begin{cases} \bigwedge\limits_{i \in LN(seq)} (\bigwedge\limits_{g \in TBEU(seq\uparrow_i)} \alpha_g) \wedge \bigwedge\limits_{j \in MSG(seq)} \beta_j \wedge \bigwedge\limits_{CF \in nested(seq)} \Phi^{CF} & (1) \\ \Phi_{seq}^{Basic} & (2) \end{cases}$$

Fig. 33. LTL templates for Sequence Diagram

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

Fig. 34. LTL formula for Interaction Use

any valid trace. We define a temporal logic template, Ω_{seq}^{NCF}, to characterize the safety property of Sequence Diagram *seq* with respect to an NCF. Formally,

$$\Omega_{seq}^{NCF} = (\neg \Phi_{NCF}) \vee (\neg \delta_{(TAOS(NCF),(TAOS(seq) \setminus TAOS(NCF)))})$$

in which formula $\delta_{(TAOS(NCF),(TAOS(seq) \setminus TAOS(NCF)))}$ asserts that the traces enclosed in NCF are not interleaved by other OSs in Sequence Diagram *seq* formula Φ_{NCF} specifies the set of invalid traces expressed by an NCF. Formula Ω_{seq}^{NCF} asserts that Sequence Diagram *seq* is safe if either of the following two conditions is satisfied. 1. Any trace of Sequence Diagram *seq* does not contain an invalid trace specified by Φ_{NCF} as a sub-trace (first disjunct). 2. Any trace in NCF is interleaved by the other OSs not in NCF (second disjunct).

Figure 35 shows an example of an NCF, which we want to check against the Sequence Diagram, *seq*, shown in figure 4. Our techniques and tools translate *seq* into NuMSV modules and define a safety property Ω_{seq}^{NCF} with respect to the example NCF. The model checker asserts that the property is satisfied in that invalid traces [*s1, r1, s3, r3*], [*s1, s3, r1, r3*], and [*s1, s3, r3, r1*] are not shown as sub-traces in the example Sequence Diagram.

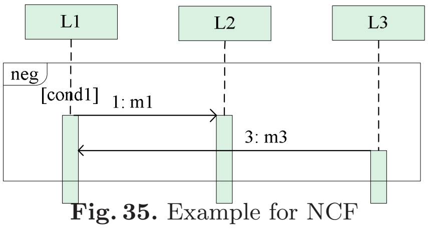

Fig. 35. Example for NCF

6.2 Consistency Property with Assertion Combined Fragment

We define that a collection of Sequence Diagrams is consistent with respect to a Sequence Diagram with an ASCF only if any trace in the ASCF on a Lifeline always follows OSs, which may happen right before the ASCF on the same Lifeline. Formula Ω_{seq}^{ASCF} in figure 37 represents the consistency property of *seq* with respect to an $ASCF$. Formula $\lambda_{N_1,N_2}^{i,seq}$ represents that on Lifeline i if all the elements in N_1 happen, no other OSs in Sequence Diagram *seq* are allowed to happen until all the elements in N_2 complete their execution.

Based on this consistency definition, we can model check if a Sequence Diagram, *seq*, satisfies the consistency constraints set by another Sequence Diagram with an ASCF. For example, we can model check Ω_{seq}^{ASCF} for the Sequence Diagram in figure 36 against the NuSMV model for *seq*, in figure 1. The consistency 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.

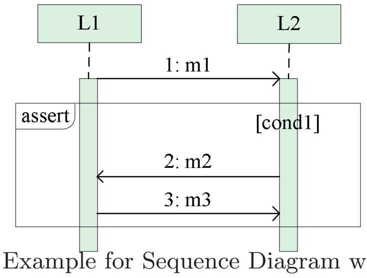

Fig. 36. Example for Sequence Diagram with ASCF

$$\lambda_{N_1,N_2}^{i,seq} = \Box\Big(\bigwedge_{OS_p \in N_1} OS_p \rightarrow \Big(\bigwedge_{OS_q \in (TAOS(seq\uparrow_i)\setminus N_2)} (OS_q \leftrightarrow \bigcirc(OS_q))\widetilde{\mathcal{U}}\Big(\bigwedge_{OS_r \in N_2} OS_r\Big)\Big)\Big)$$

$$\Omega_{seq}^{ASCF} = \bigwedge_{i \in LN(ASCF)} \lambda_{(pre(ASCF\uparrow_i),TAOS(ASCF\uparrow_i))}^{i,seq} \wedge \Phi_{ASCF}$$

Fig. 37. LTL formula for Sequence Diagram with ASCF

6.3 Deadlock Property with Synchronous Messages

The deadlock-free property can be checked in a Sequence Diagram with synchronous Messages. Deadlock can occur if multiple Lifelines are blocked, waiting on each other for a reply.

To represent a Sequence Diagram with synchronous Messages as a NuSMV model, a boolean variable, “isBlock” is introduced for each Lifeline, indicating that the sending Lifeline is blocked until a reply is received. The variable “isBlock” is considered a condition for all the OSs on the same Lifeline, preventing other OSs from executing. Initially $False$, it is set to $True$ when a Lifeline sends a Message. “isBlock” returns to $False$ when the reply arrives and the execution of other OSs resumes. Synchronous Message OS names are prefixed with either s for a synchronous Message, or r for a reply Message. Figure 38 is an example of a Sequence Diagram with synchronous Messages and Figure 39 represents its NuSMV description for Lifeline $L1$.

Figure 40 represents the LTL formula of a basic Sequence Diagram with synchronous Messages. It conjuncts some constraints with the LTL formula of basic Sequence Diagram with asynchronous Messages. We define some helper functions. $OSpre(m_j, L_i)$ returns the OS prior to Message m_j on Lifeline L_i. $syncSend$ returns the OS which is the sending OS of a synchronous Message and $replyReceive$ returns the OS which is the receiving OS of a reply Message. π_{sync} describes that a Lifeline can not send any synchronous Message until the previous synchronous Message receives its reply. In the example of figure 38, all of the Lifelines eventually deadlock since they all send Messages and are all awaiting replies. The LTL formula is not satisfied when checking against the NuSMV module.

7 Framework and Evaluation

As a proof-of-concept, we have developed a tool suite, implementing the techniques described in this paper. Figure 41 illustrates the architecture of our software framework.

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

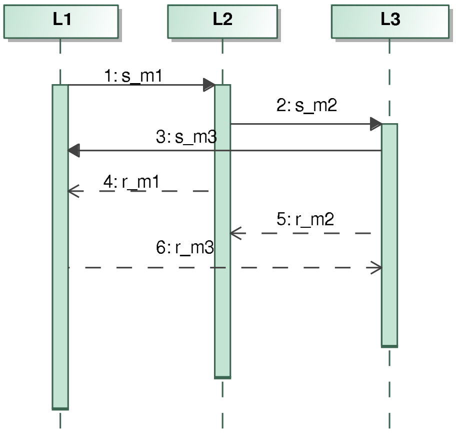

Fig. 38. Example for basic Sequence Diagram with synchronous Messages

```
MODULE L1 (L2, L3)
  DEFINE
    flag_final := OS_r_s3;
  VAR
    OS_s_s1:boolean;
    OS_r_r1:boolean;
    OS_s_r3:boolean;
    OS_r_s3:boolean;
    isblock:boolean;
  ASSIGN
    init(OS_s_s1) := FALSE;
    next(OS_s_s1) := case
      !OS_s_s1 & !isblock            :TRUE;
      1                              :OS_s_s1;
    esac;
    init(OS_s_r3) := FALSE;
    next(OS_s_r3) := case
      OS_s_s1 & L3.OS_s_s3
              & !isblock             :TRUE;
      1                              :OS_s_r3;
    esac;
    ...
    init(isblock) :=FALSE;
    next(isblock) := case
      next(OS_s_s1) & !next(OS_r_r1):TRUE;
      next(OS_r_r1)                  :FALSE;
      1                              :isblock;
    esac;
  FAIRNESS
    running;
```

Fig. 39. NuSMV Module for Sequence Diagram with synchronous Messages

$$\Psi_{seq}^{sync} = \Phi_{seq}^{Basic} \wedge \pi_{sync}$$

$$\pi_{sync} = \bigwedge_{\substack{\{m_j|typeOS(m_j,L_i)=send \\ \wedge KindM(m_j)=sync\}}} (\neg OS(m_j, L_i)\ \ \mathcal{U}\ \ replyReceive(syncSend(OSpre(m_j, L_i))))$$

Fig. 40. LTL formula for Sequence Diagram with synchronous Messages

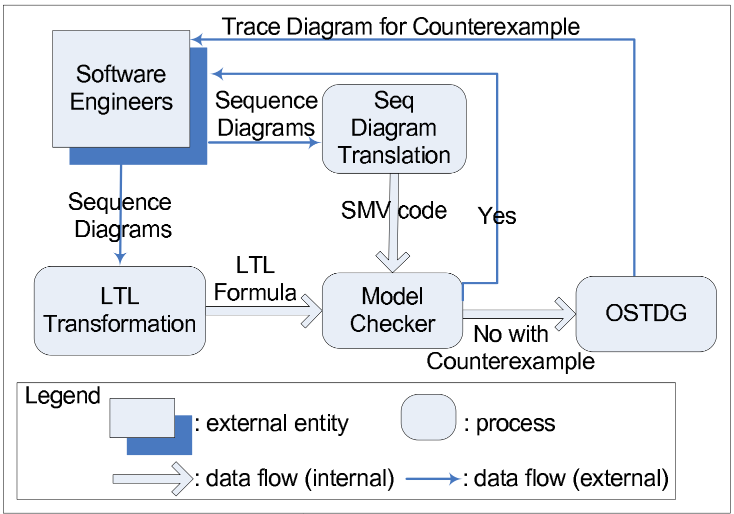

Fig. 41. Formal Analysis of Sequence Diagram

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.

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 42.

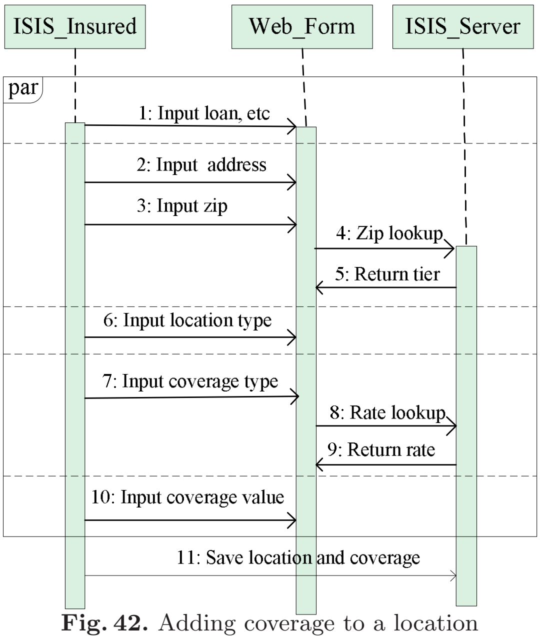

Fig. 42. Adding coverage to a location

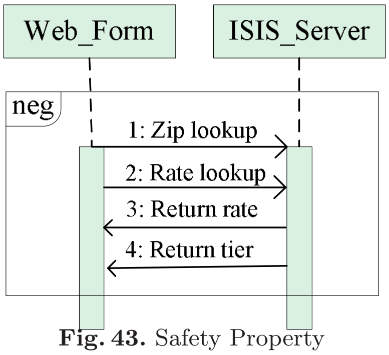

Fig. 43. Safety Property

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. Multiple users and days may be involved during the procedure. During this time, the client must be free to continue to use ISIS between the end-of-month processing and reporting tasks. However, if end-of-month reporting occurs before the billing data is sealed, the reports may contain inaccurate data and be inconsistent with 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 44. 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 43). 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 46), we easily located the messages involved in the property violation. In reality, locating this bug manually without our automtic technique involved a great deal more time and effort. Model checking the consistency property of a Sequence Diagram with ASCF (see [23] for the diagram)

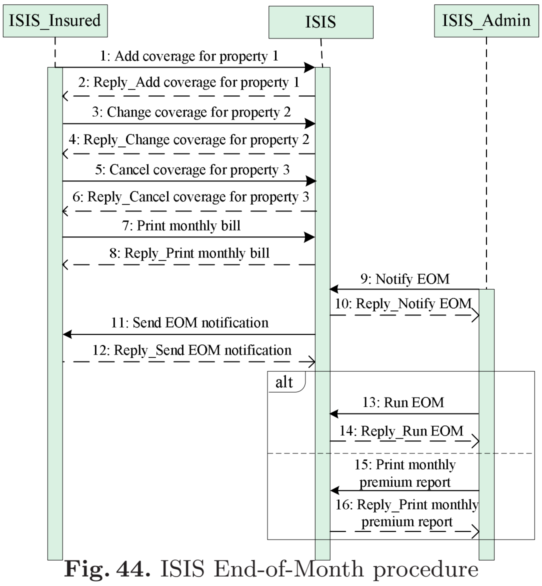
ISIS_Insured
ISIS
ISIS_Admin
1: Add coverage for property 1
2: Reply_Add coverage for property 1
3: Change coverage for property 2
4: Reply_Change coverage for property 2
5: Cancel coverage for property 3
6: Reply_Cancel coverage for property 3
7: Print monthly bill
8: Reply_Print monthly bill
9: Notify EOM
10: Reply_Notify EOM
11: Send EOM notification
12: Reply_Send EOM notification
alt
13: Run EOM
14: Reply_Run EOM
15: Print monthly premium report
16: Reply_Print monthly premium report

Fig. 44. ISIS End-of-Month procedure

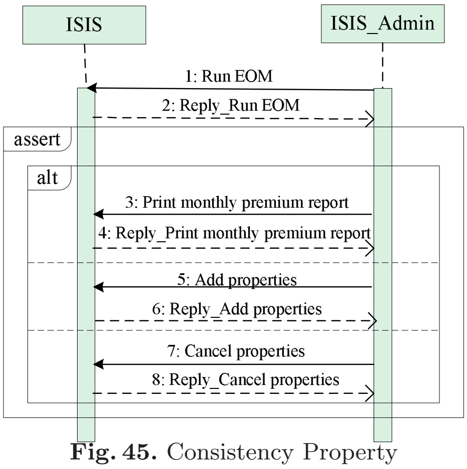

Fig. 45. Consistency Property

against example 2's model returned true, indicating that end-of-month processing is always followed by end-of-month reporting.

We use 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.

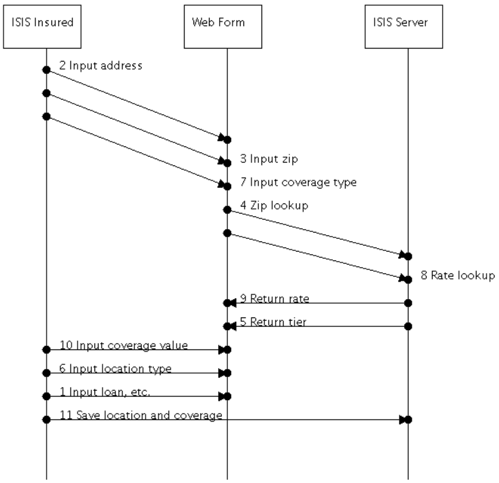

Fig. 46. Visualization for the counterexample of case study 1

8 Related Work

Micskei and Waeselynck survey comprehensively formal semantics proposed for Sequence Diagrams by 13 groups and present the different options taken in [16]. Whittle presents a three-level notation with formal syntax and semantics for specifying Use Cases and scenarios in [26]. The syntax of Interaction level follows the UML 2.0 Interaction and the semantics includes some Combined Fragment operators. Whittle and Jayaraman present an algorithm for synthesizing well-structured hierarchical state machines from scenarios with a three-level notation in [27]. The generated hierarchical state machines are used to simulate scenarios and improve readability. Our work focuses only at the level of Sequence Diagrams but formalizes a Sequence Diagram in order to verify properties. Working towards the similar goal, [9] presents an operational semantics of basic Interaction and some Combined Fragments for a translation of Interaction into automata. The automata is used to model check the communication produced by UML state machines with SPIN or UPPAAL. Uchitel et al. [24] synthesize a behavioral specification in the form of a Finite Sequential Process, which can be checked using their the labeled transition system analyzer. Bontemps et al. formally study the problem of scenario checking, synthesis, and verification of LSC in [5]. Their work focuses on providing an algorithm for each problem and proving the complexity. A comprehensive survey of these synthesis approaches and others' work can be found in [14]. Our work is different in that we formalize all the Combined Fragments in both LTL and FSMs, which promotes automated verification of Sequence Diagrams.

Verification of scenario-based notation is well-accepted as an important and challenging problem. Lima et al. provide a tool to translate UML 2 Sequence Diagrams into PROMELA-based models and verify using SPIN, with counterexample visualizations [15]. They are able to track the execution state, assisting LTL property specification, but their translation does not include all of the UML 2 Combined Fragments. Mitchell [17] demonstrates that there is a unique minimal generalization of a race-free partial-order scenario even if it is iterative. He [18] also extends the Mauw and Reniers' algebraic semantics for MSCs to describe UML 2 sequence diagrams. We use a different definition of the "deadlock" property (that at least one Lifeline does not reach its final OS in some trace). Alur et al. examine different cases of MSC verification of temporal properties and present techniques for iteratively specifying requirements [1]. They focus on MSC Graphs (an aggregation of MSCs) and techniques for determining if a particular MSC is realized in an MSC Graph. We extend their work to encompass more complicated aggregations using Combined Fragments. The group of Peled has performed intensive research on the verification of MSCs [19, 8], in particular, [21] present an extension of the High-Level Message Sequence Chart (HMSC). They specify MSC properties in temporal logic and check for safety and liveness properties. As UML 2 Sequence Diagrams have similar expressive features, our technique can be extended to work with their approach. Kugler et al. improve the the smart play-out, which is used to model check Live Sequence Charts (LSCs) to avoid violations over computations [11]. They can detect deadlock of dependent moves. Our technique can check for desired properties, such as deadlock, safety, of more complex scenario-based notation permitting various control flows. Walkinshaw and Bogdanov [25] detail an inference technique to constrain a finite-state model with linear temporal logic (LTL). These constraints reduce the number of traces required as input to a model checker for discovery of safety counter examples. Our work is similar in that we automatically translate models and properties produced by Sequence Diagrams and then verify model against specified LTL properties. Lamsweerde et.al. [13] develop an approach to inferring system properties, in terms of temporal logic, from both positive and negative scenarios. But, they only consider simple scenarios without more complicated control constructs, like Combined Fragments. Balancing flexibility and simplicity in expressing temporal properties, Autili et al.[2, 3] propose a scenario-based visual language, the Property Sequence Chart(PSC) that can be used to formalize Sequence Diagrams to permit verification.

Inconsistency among design models in different UML notations can be quite problematic on large software development projects where many developers design the same software together. Blanc et al. [4] address the problem of inconsistency between multiple Use Case and requirements models by checking model construction operations against logical inconsistency rules. They check only for consistency problems, excluding safety and deadlock problems. Also, predicate logic does not easily represent temporal qualities which we feel are necessary for accurately expressing the full range of Combined Fragments. Egyed developed approaches [7] to detect and fix 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 the more complicated features like Combined Fragments.

9 Conclusion

In this paper, we present a novel approach to formalize Sequence Diagrams and CFs into NuSMV models and LTL formulas. This enables software practitioners to verify if a Sequence Diagram satisfies specified properties and if a set of Sequence Diagrams are safe and consistent. We supplement our technique with a proof-of-concept automated tool suite and perform an evaluation using a case study of an industry web application. We believe our approach can be adapted to define the semantics of and model check other scenario-based languages.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. TSE 31(9) (2003)
2. Autili, M., Inverardi, P., Pelliccione, P.: A scenario based notation for specifying temporal properties. In: SCESM. pp. 21–28 (2006)
3. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Autom Softw Eng 14(3), 293–340 (2007)
4. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE. pp. 511–520 (2008)
5. Bontemps, Y., Heymans, P., Schobbens, P.Y.: From Live Sequence Charts to state machines and back: A guided tour. TSE 31(12), 999–1014 (2005)
6. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. Int. Journal on Software Tools for Technology Transfer 2, 410–425 (2000)
7. Egyed, A.: Instant consistency checking for the UML. In: ICSE. pp. 381–390 (2006)
8. Gunter, E.L., Muscholl, A., Peled, D.: Compositional Message Sequence Charts. In: TACAS. vol. 2031, pp. 496 – 511 (2001)
9. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: MODELS. pp. 42–51 (2006)
10. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: TACAS. pp. 445–460 (2005)
11. Kugler, H., Plock, C., Pnueli, A.: Controller synthesis from LSC requirements. In: FASE. pp. 79–93 (2009)
12. Kumar, R., Mercer, E.G., Bunker, A.: Improving translation of Live Sequence Charts to temporal logic. Electron. Notes Theor. Comput. Sci. 250(1), 137–152 (2009)
13. van Lamsweerde, A., Willemet, L.: Inferring declarative requirements specifications from operational scenarios. TSE 24(12) (1998)
14. Liang, H., Dingel, J., Diskin, Z.: A comparative survey of scenario-based to state-based model synthesis approaches. In: SCESM. pp. 5–11 (2006)
15. Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., Pourzandi, M.: Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. Electron. Notes Theor. Comput. Sci. 254, 143–160 (2009)
16. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: a survey. Software and Systems Modeling, to appear
17. Mitchell, B.: Resolving race conditions in asynchronous partial order scenarios. TSE 31(9), 767–784 (2005)
18. Mitchell, B.: Characterizing communication channel deadlocks in sequence diagrams. TSE 34(3), 305–320 (2008)
19. Muscholl, A., Peled, D.: Deciding properties of Message Sequence Charts. In: the First Int. Conf. on Foundations of Soft. Science and Comp. Structure, LNCS. vol. 1378, pp. 226 – 242 (1998)

20. Object Management Group: Unified Modelling Language (Superstructure), v2.3, 2010. Internet: www.omg.org
21. Peled, D.: Specification and verification of Message Sequence Charts. In: FORTE/PSTV. pp. 139 – 154 (2000)
22. Pnueli, A.: The temporal logic of programs. In: FOCS. vol. 526, pp. 46–67 (1977)
23. Shen, H., Robinson, M., Niu, J.: Formal analysis of scenario aggregation. Tech. Rep. CS-TR-2010-03, UTSA (2010)
24. Uchitel, S., Kramer, J., Magge, J.: Synthesis of behavioral models from scenarios. TSE 29(2), 99–115 (February 2003)
25. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: ASE. pp. 248–257 (2008)
26. Whittle, J.: Precise specification of use case scenarios. In: FASE. pp. 170–184 (2007)
27. Whittle, J., Jayaraman, P.K.: Synthesizing hierarchical state machines from expressive scenario descriptions. TOSEM 19(3), 1–45 (2010)