A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams

Faiz UL Muram, Huy Tran and Uwe Zdun
University of Vienna, Faculty of Computer Science,
Software Architecture Research Group, Vienna, Austria
Email: faiz.ulmuram|huy.tran|uwe.zdun@univie.ac.at

***Abstract*—The main challenge in software development process is to detect and fix the deviations of system's behaviors at different abstraction levels in early phases. For this purpose, UML 2 sequence diagrams are widely used for describing and analyzing the communication behavior of software systems. This paper describes a *containment checking* approach for UML 2 sequence diagrams to verify whether the behavior (or functions) described by a low-level model conforms those specified in the high-level counterpart based on model checking techniques, in order to improve the system's quality. However, creating consistency constraints and formal specifications for the sequence diagrams is a labor-intensive and error prone task. To alleviate this issue, we propose an automated transformation of sequence diagrams into formal specifications and consistency constraints that enable us to leverage the analytical powers of model checking to automatically verify the containment relationship. In addition, our approach provides the stakeholders more informative and comprehensive feedbacks regarding the inconsistency issues, and therefore helps them to efficiently identify and resolve the problems. The approach is implemented and validated using three realistic scenarios.**

***Keywords*-Containment checking; consistency checking; behavior model; sequence diagram; UML**

I. Introduction

In software development process, scenarios are often modeled using sequence diagrams to describe the interactions among environment (e.g., human beings) and components (aka lifelines) for analyzing the behavior of software systems. In the course of software system modeling, as models are created and evolved independently by different stakeholders, inconsistencies among models often occur. Therefore, it is crucial to detect and fix the inconsistencies at early phases of the software development process.

It is a general consensus that model checking techniques are best applied in the early phases of the software development process, as the costs are relatively low and the potential benefits are high. However, these techniques require formal specifications and consistency constraints of the models. It is a challenging task to accurately and correctly express such formal specifications and consistency constraints due to the substantial amount of knowledge and specialized training required not only for the formal verification technique but the formal specification language (notation and semantics), which is unusual for software architects and/or developers. Moreover, the results produced by existing model checkers (e.g., counterexamples) are rather cryptic and verbose, and therefore, they are difficult for the stakeholders, who–as mentioned above–often have limited knowledge of the underlying formal techniques, to interpret and understand [1].

To address the aforementioned problems, we developed a technique which allows to automatically check containment consistency of UML 2 sequence diagrams based on model checking techniques that has not been addressed adequately so far. The idea behind containment checking is to verify whether the behavior (or functions) described by the refined and extended low-level sequence model conforms those specified in the high-level counterpart. It allows the stakeholders to improve the quality of the complex systems by determining and resolving the deviations at design phase, before the systems are actually implemented and deployed.

Although, some semantics have been proposed for the verification of sequence diagrams against safety properties such as deadlock freedom [2]–[4], they do not cover the containment relationship between sequence diagrams. In this paper we therefore provide the efficient and simpler formalizations of sequence diagrams to track the execution state of an interaction involving send/receive events of messages and combined fragments without compromising the containment relationship. Specifically, we introduced a fully automated technique to translate both high-level and low-level sequence diagrams into temporal logic based constraints (LTL) [5] and formal behavior specifications (i.e., symbolic model language (SMV) [6]), respectively. The model checker NuSMV is used to verify the containment relationship. If the consistency constraints do not satisfy formal specifications then the model checker produces a counterexample as a trace of states. In order to facilitate better feedback, we integrate the counterexample analysis method for locating the cause(s) of inconsistency and presenting the appropriate suggestions to aid stakeholders in resolving containment inconsistencies. We have developed a tool to implement all of the techniques and have validated our approach by detecting and interpreting inconsistencies in three realistic scenarios.

The paper is structured as follows. Section II discusses the background information. In Section III we discuss our model checking based approach for containment checking in detail. In Section IV a realistic use case scenario is described in detail to illustrate our approach along with a performance evaluation of three realistic scenarios. Finally, Section V presents related work and Section VI concludes the paper.

II. Preliminaries

A. Linear Temporal Logic

In our study we use linear temporal logic (LTL) [5] for specifying the temporal relationships between the involved elements of UML sequence diagram. LTL is a prominent formalism that is highly expressive and widely used in formal verification tools. Formulas in LTL are usually constructed from boolean connectives and temporal modalities. In the mapping scheme of constructs of UML sequence diagrams into LTL, we opt for a syntactical definition of a well-formed LTL formula φ in terms of the following BNF grammar (note that p is a primitive proposition).

$$\varphi ::= \top | \bot | p | \neg\varphi | \varphi \wedge \varphi | \varphi \vee \varphi | \varphi \rightarrow \varphi | \mathbf{F}\varphi | \mathbf{X}\varphi | \mathbf{G}\varphi | \varphi \mathbf{U} \varphi | \varphi \mathbf{R} \varphi$$

Each interaction of a UML sequence diagram in the execution path is considered a primitive proposition. Temporal operators such as **F** (future), **X** (next), **G** (globally/always), **U** (until), and **R** (release) are used to represent the temporal relationships [6]. For the sake of readability, we use the logical exclusive OR operator "xor" which is not part of the traditional LTL definition but often supported by several tools.

B. NuSMV Overview

The NuSMV model checker [6] supports symbolic model verification and is widely used both in academia and industry. The language that underpins the formal specifications is the SMV specification language. Here, we only discuss the syntax elements we actually use in our mapping scheme.

The SMV specification consists of one main module, and a set of state variables and predicates on these variables. In general, our approach creates a state variable of type `boolean` in the SMV specification for the constructs of a UML sequence diagram using the keyword `VAR`. The present states and next states (i.e., predicates) of the variables are declared under the keyword `ASSIGN`. In particular, `init()`–for defining the initial state of a variable–and `next()`–for illustrating the transition to the next state. Inside the next expression of the variable, a `case...esac` expression is created for every state that lists all possible subsequent states. Normally a state variable will be initialized with `FALSE`. It can move to a different state (e.g., `TRUE`) if the incoming guard conditions are satisfied. The incoming guard conditions can contain a guard expression and/or the finishing of preceding interaction. The interaction's state will be switched back to `FALSE` after finishing its execution.

III. Containment Checking Approach for UML Sequence Diagrams

In this section, we describe our model checking based approach for addressing the problem of containment checking of UML 2 sequence diagrams. An overview of our approach is presented in Figure 1, consists of the following steps: (i) mapping the high-level sequence diagram into formal consistency constraints (i.e., LTL formulas), (ii) translating the low-level sequence diagram into formal SMV specifications,

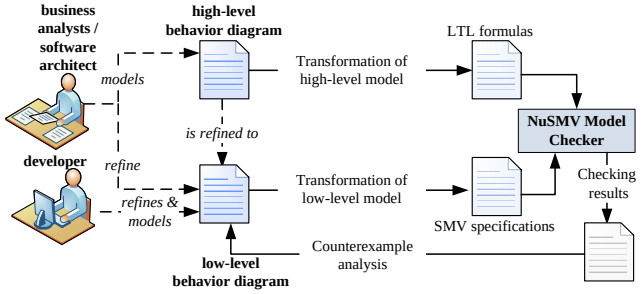


Fig. 1: Overview of the containment checking approach

(iii) verifying whether the generated constraints and specifications satisfy the containment relationship using the NuSMV model checker [7]. If the containment inconsistencies exist then generated counterexample is scrutinized to uncover the causes of inconsistencies and produce appropriate suggestions to address the deviations. The subsequent sections describe the steps involved in our containment checking approach.

A. Automated Transformation of Sequence Diagrams into LTL and SMV Specifications

As the definitions and semantics of UML 2 sequence diagrams are rather informal and ambiguous [8], we facilitate the automated creation of the formal constrains and specifications by defining transformation rules for formally representing constructs of sequence diagrams based on containment relationship. The main objective is to represent the high-level diagram's constructs and their relationships in an LTL formalism such that the execution order of the interactions will become the consistency constraints for the corresponding low-level model. Furthermore, the encoding of the low-level sequence diagram in terms of the SMV specification language should provide the foundation to facilitate the verification of the containment relationship.

Algorithm 1 Mapping UML Sequence Diagram *SeqD* into SMV Specifications / LTL

```
1:  procedure TRANSLATE(SeqD)
2:      Q ← ∅                    ▷ Q is the queue of non-visited interactions
3:      V ← ∅                    ▷ V is the queue of visited interactions
4:      Q ← Q ∪ get_lifelines(Lf)
5:      for all i ∈ Q do
6:          V ← V ∪ {i}
7:          Q ← Q \ {i}
8:          generate_smv_code(i)/generate_ltl(i)
9:          I_interactions ← get_interactions(i)
10:         for all e ∈ I_interactions do
11:             if (e ∉ V) then
12:                 Q ← Q ∪ {e}
```

Our approach takes advantage of the standards that are widely used in industry, such as Eclipse Papyrus[1], which is an Eclipse based open source UML 2 tool. In order to translate the high-level and low-level sequence diagrams into LTL formulas and SMV specifications, respectively, we leverage the Eclipse

[1] See https://www.eclipse.org/papyrus

Xtend framework[2]. We achieve the mapping of sequence diagram into SMV specifications and LTL formulas using an extended version of the breadth-first search algorithm as shown in Algorithm 1. In this algorithm, we develop four helper functions, namely, `get_lifelines()`, `get_interactions()`, `generate_smv_code()` and `generate_ltl()`. The function `get_lifelines(Lf)` returns a set of lifelines. The function `get_interactions(i)` extract all interactions *i*, i.e., messages along with sending and receiving *OccurrenceSpecifications* (*OSs*) covered by lifelines in temporal order, associated *CombinedFragments* included operands. An interaction *e* is called "receiving event" of *i* "sending event" if there is a link from *i* to *e* that synchronize with the appropriate sending and receiving lifelines.

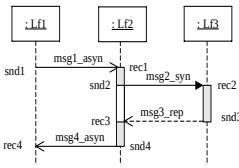
Algorithm 2 Generating SMV Specifications for an Interaction *i* of a UML Sequence Diagram *SeqD*

```
1:  procedure GENERATE_SMV_CODE(i);
2:      extracts interaction information;
3:      binds input values and generates SMV specifi-
    cations using the following templates:
4:      '''
5:      VAR
6:          «i» : boolean;          ▷ State variable declaration
7:      ASSIGN
8:          init(«i») := «interaction-initial-state»
9:          next(«i») := case
10:             «incoming-condition(s)» : TRUE;
11:             «i» : FALSE;
12:         esac;
13:     '''
```

The most important function is `generate_smv_code(i)`, which is responsible for generating SMV specifications for each interaction of a UML sequence diagram. We illustrate the skeleton of the function `generate_smv_code(i)` in Algorithm 2. The pair of triple apostrophes (''') denotes the string templates used for generating code in the our implementation. For the sake of readability, we opt to omit the verbosity of the transformation code realized using the Xtend language and use a pair of guillemots i.e., "«" and "»" to denote the parameterized placeholders that will be bound to and substituted with the actual values extracted from the low-level input model interactions by the Xtend engine, as we see in Figure 2c. In our formalizations, we map a message and its sending and receiving *OSs* (i.e., events) as a tuple $\langle Lf, msg, snd/rec \rangle$, where Lf represents the lifeline that is responsible for sending or receiving messages. *msg* denotes the message name. *snd* and *rec* represent the sending *OS* and receiving *OS* of the corresponding message on a lifeline, respectively. To represent multiple inputs for an element (i.e., state or event) the logical *AND* operator ("&") is used.

We note that `generate_smv_code(i)` is not realized as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction *i*, a particular function for generating SMV specifications for that *OS* or fragment type will be invoked. This can be achieved in traditional programming languages by using a typical "`if/then/else`" or "`switch/case`" construct. In our prototypical implementation, we leverage the powerful polymorphic method invocation technique provided by Xtend[3], which is used to realize the transformation of UML sequence diagram to SMV specifications. Using this technique, we devise multiple functions for generating SMV specifications with respect to the input types. Due to limitations of space we do not discuss `generate_ltl(i)` function here.

1) Basic Sequence Diagram: A sequence diagram without a *CombinedFragment* is referred to as a basic sequence diagram (such as the one in Figure 2a). The following rules for sending and receiving messages must be considered as the semantics of a basic sequence diagram.



(a) Basic sequence diagram

```
1. G (Lf1 -> F (
     Lf1_msg1_asyn_snd1)
     )
2. F (Lf2_msg3_rep_rec3)
     -> ((!
     Lf2_msg3_rep_rec3 )
      U (
     Lf2_msg2_syn_snd2 &
      Lf3_msg3_rep_snd3
     & wait_rep))
3. G ((Lf2_msg3_rep_rec3
     ) & ! (wait_rep) ->
      F (
     Lf2_msg2_asyn_snd4)
     )
4. F (Lf1_msg4_asyn_rec4
     ) -> ((!
     Lf1_msg4_asyn_rec4)
      U (
     Lf2_msg4_asyn_snd4
     &
     Lf1_msg1_asyn_snd1)
     )
```

(b) LTL generation rules

```
1  VAR
2    «Lf1_msg1_asyn_snd1» : boolean;
3    «wait_rep» : boolean;
4    ...
5  ASSIGN
6    init(«Lf1») := TRUE;
7    next(«Lf1») := case
8        «Lf1» : FALSE;
9    esac;
10 --sending OS on a lifeline Lf1
11   init(«Lf1_msg1_asyn_snd1») := FALSE;
12   next(«Lf1_msg1_asyn_snd1») := case
13       «Lf1» : TRUE;
14       «Lf1_msg1_asyn_snd1» : FALSE;
15   esac;
16 --receiving OS on a lifeline Lf1
17   init(«Lf1_msg4_asyn_rec4») := FALSE;
18   next(«Lf1_msg4_asyn_rec4») := case
19       «Lf1_msg1_asyn_snd1» & «
              Lf2_msg4_asyn_snd4» : TRUE;
20       «Lf1_msg4_asyn_rec4» : FALSE;
21   esac;
22   ...
23   init(«wait_rep») := FALSE;
24   next(«wait_rep») := case
25       «Lf2_msg2_syn_snd2» : TRUE;
26       «wait_rep» : FALSE;
27   esac;
28 --sending OS after receiving reply
29   init(«Lf2_msg4_asyn_snd4») := FALSE;
30   next(«Lf2_msg4_asyn_snd4») := case
31       «Lf2_msg3_rep_rec3» & ! «
              wait_rep» : TRUE;
32       «Lf2_msg4_asyn_snd4» : FALSE;
33   esac;
```

(c) SMV generation rules

Fig. 2: Translation of basic sequence diagram

- The *OSs* on the same lifeline must occur in the same order in which they are described [8, p.505].
- "The semantics of a complete message is simply the trace $\langle sendEvent, receiveEvent \rangle$" [8, p.507]. A receiving *OS* of a message is enabled for execution if and only if the sending occurrence of the same message has already occurred [8, p.507].
- If sending and receiving *OSs* of the same message are on the same lifeline then the sending event of a message must exist before its receiving event [8, p.506].

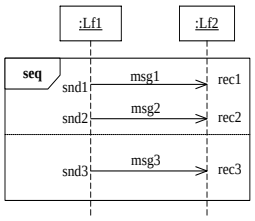
[2]See https://eclipse.org/xtend

[3]See https://eclipse.org/xtend/documentation/202_xtend_classes_members.html

Figure 2a shows a basic sequence diagram where *msg*4_*asyn* is a message received on the lifeline $Lf1$ which is sent from the lifeline $Lf2$. The receiving *OS* is only enabled when its sending *OS* (*snd*4) on $Lf2$ and its prior *OS* (*snd*1) on $Lf1$ have already occurred. The particular receiving rule is shown in Figure 2c (Line 17–21). In case the synchronous message is sent, a condition *wait_rep* is used indicating that the *OS* can not be sent when the lifeline is waiting for the reply (Line 23–33). Figure 2b shows the LTL generation rules for sending *OSs* of messages *msg*1 and *msg*4, and receiving *OSs* of messages *msg*3 and *msg*4.

2) Weak Sequencing Combined Fragment: The **seq** interaction operator imposes the order of the execution of operands associated with the same lifeline with the following constraints [8, p.483]:

- The ordering of events (i.e., *OSs*) within each of the operands are maintained.
- *OSs* on different lifelines from different operands may execute in any order.
- *OSs* on the same lifeline from different operands are ordered such that an *OS* of the first operand comes before that of the second operand.



(a) Weak Sequencing example

```
1. G (seq -> F opd1) & G
     (opd1 -> F opd2)
2. G (opd1 -> F (
     Lf1_msg1_snd1))
3. G (Lf1_msg1_snd1 -> F
     (Lf1_msg2_snd2))
4. G ((opd2 &
     Lf1_msg2_snd2) -> F
     (Lf1_msg3_snd3))
5. F (Lf2_msg3_rec3) ->
     ((! Lf2_msg3_rec3) U
      (Lf1_msg3_snd3 &
     opd2 & Lf2_msg2_rec2
     ))
```

(b) LTL generation rules

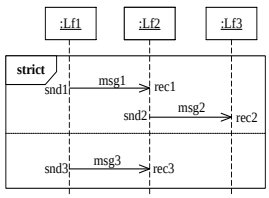
```
1  ASSIGN
2    ...
3    init(«opd1») := FALSE;
4    next(«opd1») := case
5        «seq» : TRUE;
6        «opd1» : FALSE;
7     esac;
8   --OSs in the first operand on Lf1
9    init(«Lf1_msg1_snd1») := FALSE;
10   next(«Lf1_msg1_snd1») := case
11       «opd1» : TRUE;
12       «Lf1_msg1_snd1» : FALSE;
13    esac;
14   init(«Lf1_msg2_snd2») := FALSE;
15   next(«Lf1_msg2_snd2») := case
16       «Lf1_msg1_snd1» : TRUE;
17       «Lf1_msg2_snd2» : FALSE;
18    esac;
19    init(«opd2») := FALSE;
20    next(«opd2») := case
21      «opd1» : TRUE;
22      «opd2» : FALSE;
23    esac;
24  --OS in the second operand on Lf1
25   init(«Lf1_msg3_snd3») := FALSE;
26   next(«Lf1_msg3_snd3») := case
27       «opd2» & «Lf1_msg2_snd2» :
              TRUE;
28       «Lf1_msg3_snd3» : FALSE;
29    esac;
```

(c) SMV generation rules

Fig. 3: Translation of Weak Sequencing combined fragment

Figure 3a shows an example of a *Weak Sequencing* combined fragment. Figure 3c illustrates the rules for mapping a *Weak Sequencing* combined fragment into SMV specifications. The combined fragment, corresponding operands, each of its covered lifelines, and *OSs* are mapped into state variables. The choice of the order of *OSs* is made using a "`case/esac`" construct. For instance, the sending *OS* of a message *msg*3 on a lifeline $Lf1$ within the second operand *opd*2 (i.e., `Lf1_msg3_snd3`) cannot occur until the last *OS* of the first operand *opd*1 on the same lifeline (i.e., `Lf1_msg2_snd2`) completes its execution (Line 25–29). Figure 3b shows the LTL generation rules for sending messages within both operands on a lifeline $Lf1$ (Line 2–4) and receiving a message within a second operand on a lifeline $Lf2$ (Line 5).

3) Strict Sequencing Combined Fragment: The semantics of *Strict Sequencing* (i.e., **strict** interaction operator) imposes the total order between adjacent operands. It contains a stronger version of the second rule introduced for *Weak Sequencing*, in particular, *OSs* on different lifelines from different operands have strict order of execution [8, p.483]. In other words, the first *OS* in a succeeding operand cannot be enabled until all the *OSs* on all the covered lifelines within the preceding operand have completed. Any covered lifeline needs to wait for other lifelines to enter the second or subsequent operand. For instance, sending *OS* of a message *msg*3 within the second operand covered by a lifeline $Lf1$ will not be executed until the last *OS* that is *rec*2 within a first operand on a lifeline $Lf3$ finishes its execution, as shown in Figure 4c (Line 15–19). Figure 4b shows LTL generation rules for the *Strict Sequencing*.



(a) Strict Sequencing example

```
1. G (opd1 -> F (
     Lf1_msg1_snd1))
2. F (Lf3_msg2_rec2) ->
     ((! Lf3_msg2_rec2) U
     (Lf2_msg2_snd2))
3. G ((Lf1_msg1_snd1 &
     Lf3_msg2_rec2 & opd2)
      -> F (Lf1_msg3_snd3)
     )
```

(b) LTL generation rules

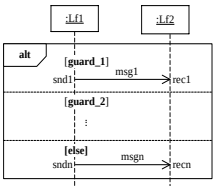
```
1  ASSIGN
2   ...--OSs within the first operand
3    init(«Lf1_msg1_snd1») := FALSE;
4    next(«Lf1_msg1_snd1») := case
5        «opd1» : TRUE;
6        «Lf1_msg1_snd1» : FALSE;
7    esac;
8   ...
9    init(«Lf3_msg2_rec2») := FALSE;
10   next(«Lf3_msg2_rec2») := case
11       «Lf2_msg2_snd2» : TRUE;
12       «Lf3_msg2_rec2» : FALSE;
13   esac;
14    --the first OS within the
          second operand
15   init(«Lf1_msg3_snd3») := FALSE;
16   next(«Lf1_msg3_snd3») := case
17       «Lf3_msg2_rec2» & «
            Lf1_msg1_snd1» & «opd2»
             : TRUE;
18       «L1_msg2_snd2» : FALSE;
19   esac;
```

(c) SMV generation rules

Fig. 4: Translation of Strict Sequencing combined fragment

4) Alternatives: In the UML 2 specification [8, p.482], an *Alternative* combined fragment describes a branching operation in a sequence diagram. The **alt** operator of the combined fragment represents a choice of behavior where at most one of the operands will be selected whose interaction constraint (guard condition) evaluates to `True` (i.e., an if-then-else statement). The **else** guard is the negation of the disjunction of all other constraints in the enclosing combined fragment. If none of the operands has a guard that evaluates to `True`, none of the operands will be executed and the remainder of the enclosing *InteractionFragment* will be performed.

Figure 5a shows an example of an *Alternative* combined fragment, whose **guard** is encoded as a boolean variable. If the **guard** of the first operand evaluates to `True`, the *OSs* enclosed within the first operand are executed, otherwise the whole operand is skipped. We introduce a temporary variable,



(a) Alternatives example

```
1. G (alt -> F ((opd1
     xor ... xor opdn)
      xor ! (opd1 &
     ... & opdn)))
2. G (opd1 & guard_1
     -> F (
     Lf1_msg1_snd1))
3. F (Lf2_msg1_rec1)
     -> ((!
     Lf2_msg1_rec1) U
     (Lf1_msg1_snd1 &
     guard_1))
4. G (opdn & else -> F
      (Lf1_msgn_sndn))
5. F (Lf2_msgn_recn)
     -> ((!
     Lf2_msgn_recn) U
     (Lf1_msgn_sndn &
     else))
```

(b) LTL generation rules

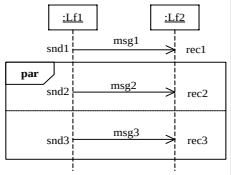
```
1  VAR
2    «guard_1» : boolean;
3    ... -- temporary variable
4    «post_alt_i» : {undetermined, «ch_opd1
         »,..., «ch_opdn»};
5  ASSIGN
6    ...
7      init(«post_alt_i») := undetermined;
8      next(«post_alt_i») := case
9          «alt» & «guard_1» : «ch_opd1»;
10         ...
11         «alt» & «else» : «ch_opdn»;
12     TRUE : undetermined;
13     esac;
14   --the first operand
15     init(«opd1») := FALSE;
16     next(«opd1») := case
17         «post_alt_i» = «ch_opd1» : TRUE;
18         «opd1» : FALSE;
19     esac;
20     init(«Lf1_msg1_snd1») := FALSE;
21     next(«Lf1_msg1_snd1») := case
22         «opd1» : TRUE;
23         «Lf1_msg1_snd1» : FALSE;
24     esac;
25   ... --the nth operand
26     init(«opdn») := FALSE;
27     next(«opdn») := case
28         «post_alt_i» = «ch_opdn» : TRUE;
29         «opdn» : FALSE;
30     esac;
31     init(«Lf1_msgn_sndn») := FALSE;
32     next(«Lf1_msgn_sndn») := case
33         «opd2» : TRUE;
34         «Lf1_msgn_sndn» : FALSE;
35     esac;
```

(c) SMV generation rules

Fig. 5: Translation of Alternative combined fragment

namely, `post_alt_i` (i is an incrementally generated number) for exclusively choosing one of many alternative operands. The variable `post_alt_i` has an enumerated type including a normal state "`undetermined`" and the values corresponding to the operands (Line 4). The choice among alternatives is made using a "`case/esac`" construct as shown in Figure 5c (Line 7–13). The LTL generation rules for the **alt** operator enumerate all possible choices of executions; that is, only *OSs* of one of the operands, whose guard evaluates to `True`, will happen, as shown in Figure 5b.

5) Parallel Combined Fragment: A *Parallel* combined fragment is denoted by an interaction operator **par** which defines potentially parallel merge execution of behaviors of the operands [8, p.483]. The *OSs* of different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved. In other words, *OSs* of messages within the same operand respect the order along a lifeline whilst *OSs* of messages on the same lifeline from different operands are ordered such that the first message occurrence of the operands has the same preceding *OS*. Figure 6c shows the translation of a *Parallel* combined fragment into SMV specifications where a sending *OS* of a message *msg*1 (`Lf1_msg1_snd1`) on a lifeline $Lf1$ leads to the execution of sending *OSs* of messages in both operands (i.e., `Lf1_msg2_snd2` and `Lf1_msg3_snd3`). LTL generation rules for *Parallel* combined fragments for the covered lifelines $Lf1$ and $Lf2$ are presented in Figure 6b.



(a) Parallel fragment

```
1. G (par -> F (opd1 &
     opd2))
2. G (Lf1_msg1_snd1 &
     opd1 -> F (
     Lf1_msg2_snd2)) &
      G (Lf1_msg1_snd1
      & opd2 -> F (
     Lf1_msg3_snd3))
3. F (Lf2_msg2_rec2) ->
      ((!
     Lf2_msg2_rec2) U
     (Lf2_msg1_rec1 &
     Lf1_msg2_snd2)) &
      F (Lf2_msg3_rec3
     ) -> ((!
     Lf2_msg3_rec3) U
     (Lf2_msg1_rec1 &
     Lf1_msg3_snd3))
```

(b) LTL generation rules

```
1  ASSIGN
2  ...
3     init(«Lf1_msg1_snd1») := TRUE;
4     next(«Lf1_msg1_snd1») := case
5         «Lf1» : TRUE;
6         «Lf1_msg1_snd1» : FALSE;
7     esac;
8  ... --the first operand on Lf1
9     init(«opd1») := TRUE;
10    next(«opd1») := case
11        «par» : TRUE;
12        «opd1» : FALSE;
13    esac;
14    init(«Lf1_msg2_snd2») := FALSE;
15    next(«Lf1_msg2_snd2») := case
16        «opd1» & «Lf1_msg1_snd1» : TRUE;
17        «Lf1_msg2_snd2» : FALSE;
18    esac;
19    --the second operand on Lf1
20       init(«opd2») := TRUE;
21       next(«opd2») := case
22           «par» : TRUE;
23           «opd2» : FALSE;
24       esac;
25       init(«Lf1_msg3_snd3») := FALSE;
26       next(«Lf1_msg3_snd3») := case
27           «opd2» & «Lf1_msg1_snd1» : TRUE;
28           «Lf1_msg3_snd3» : FALSE;
29       esac;
```

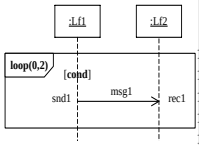
(c) SMV generation rules

Fig. 6: Translation of Parallel combined fragment

6) Loop Combined Fragment: Describing *Loop* combined fragments in terms of state-based formal specifications like SMV is a very challenging task. The interaction operator **loop** defines that its sole operand will be repeated for at-least the minimum (`minint`) number of times and at-most maximum (`maxint`) number of times as long as the guard condition remains `True` [8, p.485]. If the loop has no bounds, this means that an indefinite loop (with `minint = 0` and `maxint = infinite`) is executed, which can cause a state space explosion for model checking. However, it is unrealistic for most loops that they really execute indefinitely, and therefore, we assume that loops will eventually stop.

Figure 7a shows an example of a *Loop* combined fragment having `minint = 0` and `maxint = 2`. To deal with *Loop* combined fragments, we initialize the control variable, namely *counter*, which is `0` initially (i.e., equal to `minint`). After the end of the current iteration, the *counter* is increased by one at the beginning of the next iteration shown in Figure 7c (Line 8). Furthermore, the loop condition and *counter* are checked at the beginning of each iteration (Line 5–11). If the condition is evaluated to `False` or *counter* is greater or equals to `maxint`, a new iteration cannot start and execution of the loop will terminate. The *OSs* of all the messages within the operand among iterations execute sequentially along a lifeline. Figure 7b presents the LTL generation rules for the *Loop* fragment.

7) Option, Break, Ignore and Consider Combined Fragments: The interaction operator **opt** designates that the combined fragment represents a branching operation (i.e., a simple if-then statement) in a sequence diagram where either the (sole) operand executes or nothing happens [8, p.483]. The *Option* combined fragment is semantically similar to an *Alternative* combined fragment, except that it has one operand



(a) Loop fragment

```
1. G ((cond & counter
      < 2) -> F opd)
2. G (opd -> F (
      Lf1_msg1_snd1))
3. F (Lf2_msg1_rec1)
      -> ((!
      Lf2_msg1_rec1) U
      (Lf1_msg1_snd1))
```

(b) LTL mapping rules

```
VAR
   «counter» : «0..2»;
 ...
ASSIGN
   init(«cond») := {TRUE, FALSE};
   init(«counter»):= ((0));
   next(«counter»):= case
      «Lf2_msg1_rec1» & («counter» >= 0)
           & («counter» < 2) : «counter»
            +1;
      («counter» >= 2) : «counter»;
      TRUE : «counter»;
   esac;
   init(«opd») := FALSE;
   next(«opd») := case
      «counter» < 2 & «cond» : TRUE;
      «opd» : FALSE;
   init(«Lf1_msg1_snd1») := FALSE;
   next(«Lf1_msg1_snd1») := case
      «opd» : TRUE;
      «Lf1_msg1_snd1» : FALSE;
      esac;
   init(«Lf2_msg1_rec1») := FALSE;
   next(«Lf2_msg1_rec1») := case
      «Lf1_msg1_snd1» : TRUE;
      «Lf2_msg1_rec1» : FALSE;
      esac;
```

(c) SMV generation rules

Fig. 7: Translation of Loop combined fragment

with non-empty content and there is no **else** guard. The interaction operator **break** represents a breaking scenario in the sense that if the interaction constraint (guard condition) evaluates to `True` its sole operand executes [8, p.483]. When the constraint evaluates to `False`, the break operand will be ignored and the remainder of the enclosing *Interaction-Fragment* will be performed. The constraint after the break operand is the negation of the break operand's constraint (i.e., `!guard` is `True`). The interaction operator **ignore** defines that there is a set of messages that needs to be ignored within the combined fragment [8, p.487]. Conversely, the interaction operator **consider** specifies a set of messages that are to be considered within the combined fragment; all other messages are ignored. Due to the space limitation, the mapping rules for *Option*, *Break*, *Ignore* and *Consider* combined fragments into LTL and SMV specifications are omitted.

B. Checking Containment between Sequence Diagrams via NuSMV

The main goal of our approach is to assess whether the "execution" of the low-level model includes the "execution" prescribed in the high-level model (in the same order of executed elements). More specifically, containment checking for sequence diagrams aims to verify whether the elements and structures of a high-level sequence diagram (e.g., lifelines, sending and receiving events of messages and combined fragments) correspond to those of a refined and extended low-level sequence diagram. In our approach, containment checking is achieved by utilizing the NuSMV model checker. NuSMV takes as inputs the generated SMV specifications and LTL formulas, and exhaustively explore all executions of the SMV specifications by traversing the complete state space to determine whether the temporal logic properties hold. In case the SMV specifications satisfy the LTL formulas, this implies that the behavior described in the high-level sequence diagram is contained in the low-level sequence diagram's behavior. Otherwise, the low-level sequence diagram deviates improperly from the high-level counterpart. In this case, NuSMV will generate a counterexample that consists of the execution traces of the SMV specifications leading to the violation, which is then passed to our counterexample analysis tool. Note that the counterexample provides only limited information for understanding the causes of inconsistencies but not how to fix the inconsistencies. In this regard, the actual causes of the unsatisfied containment relationship are located based on the generated counterexamples and appropriate guidelines to resolve the particular inconsistencies are produced. Finally, the concise descriptions of the violation's causes and potential countermeasures, produced are annotated in the low-level sequence diagram.

IV. Evaluation

A. ATM System Scenario

In this section, we present a realistic scenario, namely, the Automated Teller Machine (ATM) to validate whether our proposed approach helps identifying containment inconsistencies in UML sequence diagrams. The high-level representation of the ATM system in terms of a UML sequence diagram is shown in Figure 8. For performing the containment checking first the high-level sequence diagram of the ATM system is automatically translated into LTL formulas. Afterwards, the low-level ATM system–a refined version of the high-level model is automatically converted into SMV specifications using our translation tool. Finally, the containment checking is achieved by using the NuSMV model checker.

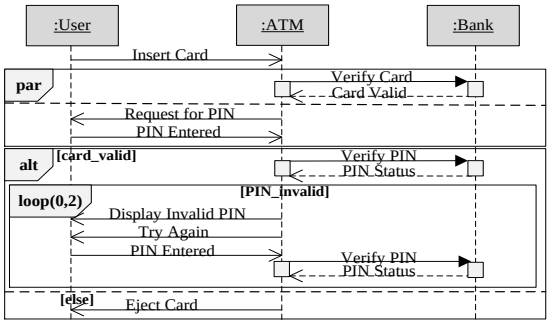


Fig. 8: High-level sequence diagram of ATM system

Listing 1 shows an excerpt of a violation trace generated by NuSMV including the list of satisfied and unsatisfied LTL formulas, i.e., a counterexample. By looking at the violation reported as a counterexample, we found that LTL formulas "`G (User_DisplayInvalidPIN_Rec8 -> F ATM_TryAgain_Snd9)`" and "`(F User_TryAgain_Rec9 -> (!User_TryAgain_Rec9 U ( ATM_TryAgain_Snd9 & User_DisplayInvalidPIN_Rec8)))`" are violated. This means that this sequence of formal properties specified by the high-level ATM system is not contained in its low-level counterpart. Despite the size and execution traces of this counterexample, the exact cause of the containment inconsistency is unclear, for instance, "is the containment

```
$ NuSMV ATM.smv
-- specification (F User_DisplayInvalidPIN_Rec8 -> (!
     User_DisplayInvalidPIN_Rec8 U ATM_DisplayInvalidPIN_Snd8)) is
     true
-- specification G (ATM_DisplayInvalidPIN_Snd8 -> F
     ATM_TryAgain_Snd9) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  User = FALSE
-- specification (F User_TryAgain_Rec9 -> (!User_TryAgain_Rec9 U (
     ATM_TryAgain_Snd9 & User_DisplayInvalidPIN_Rec8))) is false
...
```

Listing 1: NuSMV containment checking result

inconsistency caused by a missing element, or a misplacement of elements, or both of them?".

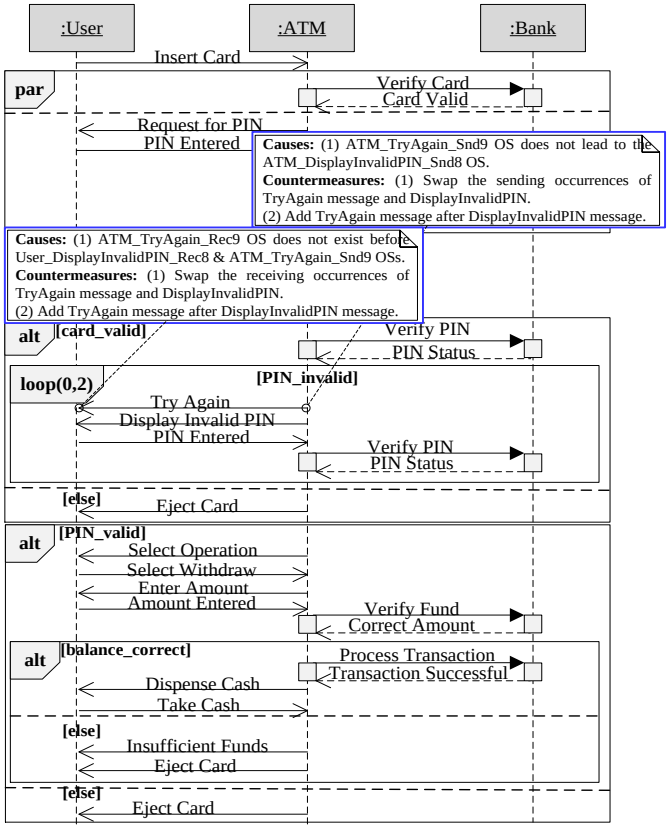


Fig. 9: Feedback of containment results in the low-level system

After the generation of the counterexample, it is important to analyze the generated counterexample to find the actual source of the inconsistency and correct the responsible elements in the sequence diagram. In order to interpret the generated counterexample, we applied our counterexample analysis technique that creates annotation at the first element causing the inconsistency to show the description of inconsistency causes and suggestions. In this case, the sending and receiving *OSs* rules for the `TryAgain` message are violated because the `TryAgain` message is sent and received prior to the receiving *OS* of the `DisplayInvalidPIN` message in the low-level model. These violations can be resolved by putting the `TryAgain` message after the `DisplayInvalidPIN` message in the low-level ATM system. In Figure 9, the blue boxes show the actual causes and potential countermeasures of unsatisfied formulas. Once the causes are located, causes are eliminated by updating the responsible elements of the low-level sequence diagram.

TABLE I: Model size and translation time

Input size	Order processing		ATM system		Itinerary management	
	HL	LL	HL	LL	HL	LL
Occurrence Specifications	20	30	26	52	34	72
Interaction Operators	2	3	3	5	4	8
Interaction Operands	5	7	5	9	7	22
Lifelines	4	4	3	3	5	6
Total Elements	31	44	37	69	50	108
Model Loading (ms)	2.344±0.25	2.680±0.49	2.515±0.41	4.223±1.28	3.374±0.57	5.126±0.65
Translation Time (ms)	0.289±0.03	0.464±0.08	0.341±0.05	0.646±0.14	0.462±0.21	0.815±0.41

B. Performance Evaluation

The main idea behind our performance evaluation is to validate whether the proposed approach provides considerable support for typical models used in real-world settings. The performance evaluation is conducted on a regular computer equipped with an 2.6 GHz i5 processor and 8GB of memory using NuSMV 2.5.4 running under Windows 8. In addition to the ATM system presented in Section IV-A, we perform the evaluation on two other industrial scenarios with different sizes and complexity, in particular, Order processing and Itinerary management, adapted from our previous projects in e-business domain [9]. The reported times include model loading, generating and verification of models measured in milliseconds. Table I shows the complexity of the input sequence diagrams (HL = high-level model, LL = low-level model) with respect to their elements including *OSs* of messages, interaction operators and operands, and covered lifelines.

TABLE II: Performance evaluation results

Containment checking	Order processing	ATM system	Itinerary management
Verification Time (ms)	127.55±10.350	1011.25±22.320	768.57±53.049
Total Time (ms)	133.327	1018.975	774.511
Violated Formulas	0 out of 22	2 out of 30	1 out of 38
Reachable States	6 (2^2.58496)	4056 (2^11.9858)	32 (2^5)
Total States	2.2518e+015 (2^51)	1.95846e+026(2^87.3399)	5.10424e+038(2^128.585)

Table II shows the total execution time of three models, reachable states and violated formulas. The evaluation results indicate that the containment checking time spent by NuSMV for the ATM system is longer than for the Itinerary management and Order processing. This is the case because NuSMV found inconsistencies between the formal specifications of the low-level model and LTL formulas of the high-level model and thus NuSMV needed to generate a counterexample for two violated LTL formulas. The evaluation results demonstrate that our approach efficiently translates sequence diagrams into formal specifications and consistency constraints for supporting containment checking. In particular, all realistic scenarios are handled in a total time around a second which is quite reasonable for practical purposes. Our analysis and evaluation results based on the aforementioned use case scenarios show the feasibility of our approach for larger realistic scenarios.

V. Related Work

Some attempts have been made to give a formal definition for UML sequence diagrams to enable model checking. For

instance, Alawneh et al. introduce a unified paradigm to verify and validate prominent diagrams, including sequence diagrams, using NuSMV [10]. The proposed semantics is not in full accordance with the standard semantics specified in UML 2 due to the lack of send and receive events. Moreover, the approach only supports alternatives and parallel combined fragments. Störrle [2] proposes the semantics for sequence diagrams in terms of the set of valid and invalid traces for "plain InteractionFragments", i.e., ones without combined fragments. Haugen et al. present the formal semantics of sequence diagram through an approach named STAIRS [3]. STAIRS focuses on the refinement for interactions, as a tuple $\langle action, sender, receiver, messagename \rangle$. However, this notation cannot describe the order of occurrences, where same message appears twice on the same lifelines. In contrast our work considers that each *OS* is unique within a sequence diagram. The aforementioned approaches have ignored the guard conditions, which compromises soundness of containment reasoning. Lima et al. provide a tool to translate sequence diagrams into PROMELA and verify using SPIN model checker [11]. Their translation does not support strict sequencing, consider and ignore combined fragments, as well as synchronous messages. Leue et al. translate the message sequence charts (MSCs), especially branching and iteration of high-level MSC into PROMELA to verify MSCs using the XSPIN tool [12]. Jacobs and Simpson present the translation of sequence diagram into process algebra CSP to investigate whether a potential design meets its specification using FDR refinement checker [4]. None of these semantics, in our opinion, achieves the simplicity and conceptual clarity of verifying the containment relationship for sequence diagrams. They are only useful for verification of sequence diagrams against safety properties such as deadlock freedom.

The work presented in this paper provides translation of both high-level and low-level sequence diagrams into LTL properties and SMV specifications. In addition, the proposed approach provides more informative and comprehensive feedbacks regarding the inconsistency issues, and thus, do not require the strong knowledge of formal methods. Our earlier works presented in [13], [14] focus on the containment checking problem for activity diagrams. In [14] we presented a graph-based approach for addressing the problem of containment checking. In another study [13] we derived a transformation of the input activity diagrams to equivalent formal specifications to enable containment checking based on model checking techniques. The main contribution of this paper concerns the automated formalization of sequence diagrams and counterexample analysis for locating the cause(s) of inconsistency problems and their resolutions.

VI. Conclusion and Future Work

This paper presented a model checking based approach to automatically detect containment inconsistencies between UML 2 sequence diagrams at different levels of abstraction in order to improve the system's quality. To this end, we proposed a translation technique for automated generation of consistency constraints (i.e., LTL formulas) and SMV specifications from high-level and low-level sequence diagrams, respectively. The NuSMV model checker is employed for verifying containment relationship. The approach also provides more informative and comprehensive feedbacks to understand and resolve the cause(s) of containment inconsistencies, and thus do not require strong background of formal techniques. In order to illustrate the applicability of the proposed approach, we realized realistic scenarios from various domains and also evaluated the performance our approach in these cases. Our research agenda includes developing support for other behavior models radically different from sequence diagrams, such as BPEL and statecharts. Another future work is to include interaction operators neg and assert.

Acknowledgment

This work is supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

References

[1] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 102–116, 2004.

[2] H. Störrle, "Trace semantics of interactions in uml 2.0," *J. Visual Languages and Computing*, 2004.

[3] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen, "Stairs towards formal design with sequence diagrams," *Software & Systems Modeling*, vol. 4, no. 4, pp. 355–357, 2005.

[4] J. Jacobs and A. Simpson, "On a process algebraic representation of sequence diagrams," in *Software Engineering and Formal Methods*. Springer International Publishing, 2015, vol. 8938, pp. 71–85.

[5] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

[6] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *11th Int'l Conf. on Computer Aided Verification (CAV)*. London, UK: Springer-Verlag, 1999, pp. 495–499.

[7] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *8th Int'l Conf. on Computer Aided Verification (CAV)*, 1996, pp. 419–427.

[8] Object Management Group, "UML 2.4.1 superstructure specification," http://www.omg.org/spec/UML/2.4.1, last accessed: 2016-06-01.

[9] H. Tran, U. Zdun, T. Holmes, E. Oberortner, E. Mulo, and S. Dustdar, "Compliance in service-oriented architectures: A model-driven and view-based approach," *Information and Software Technology*, vol. 54, no. 6, pp. 531 – 552, 2012.

[10] 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 *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, March 2006, pp. 10 pp.–418.

[11] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi, "Formal verification and validation of uml 2.0 sequence diagrams using source and destination of messages," *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 143–160, Oct. 2009.

[12] S. Leue and P. B. Ladkin, "Implementing and verifying MSC specifications using promela/xspin," in *Proc. of the DIMACS Workshop SPIN96*, 1996, pp. 65–89.

[13] F. U. Muram, H. Tran, and U. Zdun, "Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking," in *11th Int'l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*, Grenoble, France, Apr. 2014, pp. 93–107.

[14] H. Tran, F. U. Muram, and U. Zdun, "A graph-based approach for containment checking of behavior models of software systems," in *Enterprise Distributed Object Computing Conference (EDOC), 2015 IEEE 19th International*, Sept 2015, pp. 84–93.