Available online at www.sciencedirect.com
ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 254 (2009) 143–160

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages[*]

V.Lima, C. Talhi, D. Mouheb, M. Debbabi, and L. Wang [1]

Computer Security Laboratory
Concordia University
Montreal, Canada

Makan Pourzandi[2]

Software Research
Ericsson Canada
Town of Mount-Royal, Canada

Abstract

A major challenge in software development process is to advance error detection to early phases of the software life cycle. For this purpose, the Verification and Validation (V&V) of UML diagrams play a very important role in detecting flaws at the design phase. It has a distinct importance for software security, where it is crucial to detect security flaws before they can be exploited. This paper presents a formal V&V technique for one of the most popular UML diagrams: sequence diagrams. The proposed approach creates a PROMELA-based model from UML interactions expressed in sequence diagrams, and uses SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL). The whole technique is implemented as an Eclipse plugin, which hides the model-checking formalism from the user. The main contribution of this work is to provide an efficient mechanism to be able to track the execution state of an interaction, which allows designers to write relevant properties involving send/receive events and source/destination of messages using LTL. Another important contribution is the definition of the PROMELA structure that provides a precise semantics of most of the newly UML 2.0 introduced combined fragments, allowing the execution of complex interactions. Finally, we illustrate the benefits of our approach through a security-related case study in a real world scenario.

Keywords: Verification and Validation, Sequence Diagrams, Model Checking, Linea Temporal Logic, UML Design.

doi:10.1016/j.entcs.2009.09.064

1 Introduction

A major challenge in software development process is to advance error detection to early phases of the software life cycle. For this purpose, the Verification and Validation (V&V) of UML diagrams play a very important role in detecting flaws at the design phase. It has a distinct importance for software security, where it is crucial to detect security flaws before they can be exploited. *Verification* is defined as "the process of determining that a model or simulation implementation accurately represents the developer's conceptual description and specification". Whereas *validation* is defined as "the process of determining the degree to which a model or simulation is an accurate representation of the real-world from the perspective of the intended uses of the model or simulation" [21]. In this study, we focus on formal V&V of one type of UML diagrams: sequence diagrams. UML sequence diagrams are behavioral diagrams used to specify interactions among system entities in many different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation [10]. Along with class diagrams and use case diagrams, sequence diagrams are the most popular diagrams of UML [23].

During the recent years, many techniques have been proposed for V&V of UML diagrams, e.g., static analysis, theorem proving, model checking, etc. Those approaches have different strengths in different areas. Since model checkers provide automated tools for verification of a given behavioral property, they have often been used in behavioral diagrams to ensure whether the system meets the pre-defined requirements. Though, most of the proposed approaches target only activity and state machine diagrams [4, 8, 9, 12, 14, 16, 19, 20]. There are some approaches targeting sequence diagram [2, 23]. However, when it comes to interactions, it is important to analyze the type of messages being exchanged, as well as their source and destination, and their send and receive events. The proposed approaches targeting sequence diagram mainly focus on getting a formal representation of interactions, and they miss a well-defined methodology to analyze all these important elements. Moreover, those works either do not take into account UML combined fragments (components newly introduced to UML 2.0 that allow designers to describe a number of traces in a compact and concise manner [17]) or their semantics models are not in accordance with the semantics defined in the UML 2.0 specification.

[⋆] This work was possible due to funding and scientific cooperation with Ericsson Canada Software Research.

[1] Email:{v_nune,talhi,d_mouheb,debbabi,wang}@ciise.concordia.ca

[2] Email: makan.pourzandi@ericsson.com

The main contribution of this work is to provide an efficient mechanism to be able to track the execution state of an interaction, which allows designers to write relevant properties involving send/receive events and source/destination of messages using LTL. This mechanism was implemented in such a way that allows the designers to select the portion of the information that is relevant to their properties. Consequently, it gives them flexibility to write very expressive properties. Another important contribution is the definition of the PROMELA structure that provides a precise semantics of most of the newly UML 2.0 introduced combined fragments, allowing the execution of complex interactions. It allows the developer to simulate much more complex sequence diagrams, with non-straightforward execution trace. The result of these contributions is an efficient approach which is capable of detecting more flaws on more complete and complex interactions.

The proposed approach creates a PROMELA-based model from UML interactions expressed in sequence diagrams, and uses SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL). PROMELA/SPIN was chosen because it provides important concepts (sending and receiving primitives, parallel and asynchronous composition of concurrent processes, and communication channels) that are necessary to implement sequence diagrams [15]. This makes the implementation easier since the communication primitives and channels are already available in PROMELA and it does not need any extra effort to implement them. The whole technique is implemented as an Eclipse[3] plugin, which hides the model-checking formalism from the user and allows the V&V engine to be embedded into the development environment.

The remainder of this paper is structured as follows. Section 2 discusses how to get PROMELA code according to the semantics of UML 2.0 interactions. In Section 3, the mechanism for V&V using source/destination and send/receive events is presented. In Section 4, we present a scenario applying our approach, as well as experimental results. Section 5 briefly presents details about the tool implementing the V&V mechanism. In Section 6, we present the related work. Finally, we conclude with some remarks and directions for future work in Section 7.

[3] http://www.eclipse.org/

2 Translation of UML 2.0 Combined Fragments into PROMELA

In this section, we briefly present the PROMELA representation of the basic elements of sequence diagrams as defined in [15, 22]. Then we present the trace semantics of the most popular combined fragments and their respective PROMELA code that correctly simulates the execution traces. The composition of the presented translation rules allows the simulation of complex interactions with interesting and non-straightforward execution trace.

2.1 Basic Elements

The work presented in [15] specifies how to translate basic elements of MSCs into PROMELA and [2] shows that this schema can be reused for basic elements of sequence diagrams. The translation rules for basic elements presented here are based on the work proposed in those approaches, and they will be the basis for the next (and more complex) interaction elements. The PROMELA elements used for representing basic components of interactions are: (1) `proctype`: it is used for declaring new process behaviour, (2) `mtype`: it defines symbolic names of numeric constants that are used as messages in the communicating process. (3) `cham`: it declares and initializes communication channels. Finally, (4) `!/?` operators: Theses symbols are used for sending/receiving messages to/from channels, respectively.

Table 1 provides the PROMELA representation of the basic elements shown in Fig. 1. The semantics of this interaction is the single trace $\langle !m, ?m \rangle$ [10].

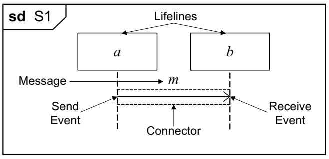


Fig. 1. Simple Sequence Diagram

2.2 Interaction Fragments and Weak Sequencing Combined Fragments

An Interaction Fragment is an abstract notion of the most general interaction unit. In other words, it is a piece of an interaction [17]. In Fig. 2(a) two messages (p and q) are sent from a to b. Each message has the semantics given for the message in Fig. 1. The vertical positions of events represent their order on each lifeline. However, the two lifelines are independent [10].

UML element	PROMELA element	PROMELA statement
Lifeline	Process	`proctype{...}`
Message	Message	`mytpe = {m1,...mn}`
Connector	Communication channel for each message arrow	`chan chanName = [1] of {mtype}`
Send and Receive events	Send and Receive operations	Send ⇒ `ab!m`, Receive ⇒ `ab?m`

Table 1
Mapping of basic UML Sequence Diagrams into PROMELA

Thus, the possible execution traces can be derived from Fig. 2(a) by using the weak sequencing operator (**seq**) defined in [17]. This operator is also used in weak sequencing combined fragment as shown in Fig. 2(b). The operator **seq** defines the set of traces with the following constraints [10, 17]:

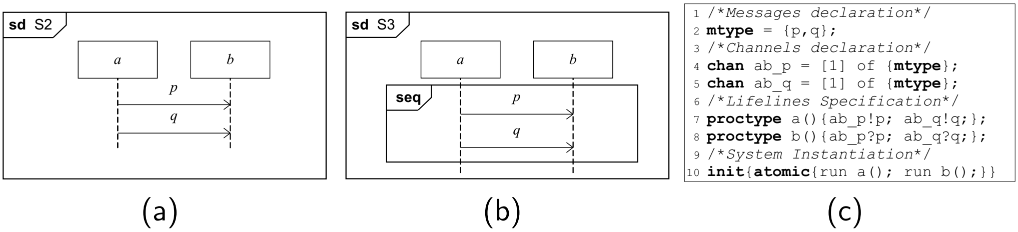
(i) The order of events within each of the operands is maintained in the result.

(ii) Events on different lifelines from different operands may come in any order.

(iii) Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.

For the interactions in Fig. 2(a) and 2(b), we get the following result:

$$S2 = \langle !p, ?p\rangle \ \mathbf{seq} \ \langle !q, ?q\rangle = \{\langle !p, ?p, !q, ?q\rangle, \langle !p, !q, ?p, ?q\rangle\} \tag{1}$$



```
1 /*Messages declaration*/
2 mtype = {p,q};
3 /*Channels declaration*/
4 chan ab_p = [1] of {mtype};
5 chan ab_q = [1] of {mtype};
6 /*Lifelines Specification*/
7 proctype a(){ab_p!p; ab_q!q;};
8 proctype b(){ab_p?p; ab_q?q;};
9 /*System Instantiation*/
10 init{atomic{run a(); run b();}}
```

Fig. 2. (a) Simple Interaction Fragment, (b) Weak Sequencing Combined Fragment and (c) their corresponding PROMELA Code

2.2.1 *PROMELA Representation:*

The communication primitives available in PROMELA naturally implements the **seq** operator following the translation map shown in table 1. This is one of the main reasons for choosing PROMELA/SPIN for model checking of sequence diagrams. Fig 2(c) shows the PROMELA code for the interactions in Fig. 2(a) and Fig. 2(b). In line 2 the messages are declared, lines 4 and 5 represent the channels on which the messages are sent, lines 7 and 8 specify the lifelines using process, and line 10 is the instruction to instantiate the system.

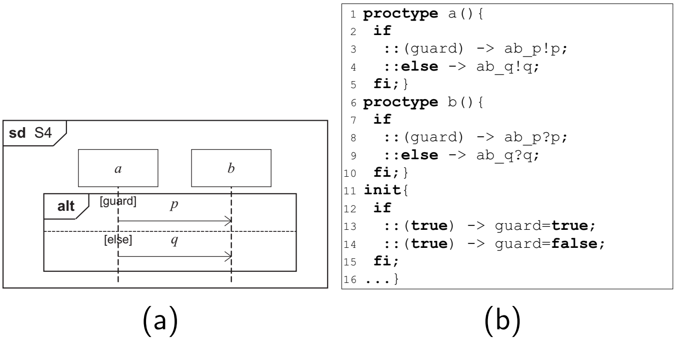
2.3 Alternative and Option Combined Fragments

Alternative and Option combined fragments represent a choice of behaviour in sequence diagrams. Alternative and Option operators are denoted as **alt** and **opt**, respectively [17]. The **opt** operator designates that the combined fragment represents a behaviour choice where either the sole operand happens or nothing happens. An option is semantically equivalent to an alternative combined fragment where there is one non-empty operand and the second operand is empty [17]. The set of traces that defines a choice is the union of the traces of the operands [10, 17]. Eq. 2 shows the set of traces of the interaction in Fig. 3(a).

$$S4 = \langle !p, ?p\rangle \ \mathbf{alt}\ \langle !q, ?q\rangle = \{\langle !p, ?p\rangle, \langle !q, ?q\rangle\} \qquad (2)$$

2.3.1 PROMELA Representation:

Alternative and Option operator are represented as *if* condition in PROMELA. The guard variable is declared globally to enforce all lifelines to get the same decision at the choice point. The non-deterministic behaviour is implemented at the set-up time by assigning different values to the guards using *if* statement with two executable conditions (lines 13 and 14 of Fig. 3(b)). At execution time, SPIN randomly chooses an option and continues the simulation. In exhaustive mode, SPIN will simulate all possible system decisions and it will provide all traces shown in Eq. 2. Fig. 3(b) presents the PROMELA code corresponding to the model in Fig. 3(a).



```
1 proctype a(){
2  if
3    ::(guard) -> ab_p!p;
4    ::else -> ab_q!q;
5  fi;}
6 proctype b(){
7  if
8    ::(guard) -> ab_p?p;
9    ::else -> ab_q?q;
10 fi;}
11 init{
12 if
13   ::(true) -> guard=true;
14   ::(true) -> guard=false;
15 fi;
16 ...}
```

(a) (b)

Fig. 3. (a) Alternative Combined Fragment, (b) Respective PROMELA code

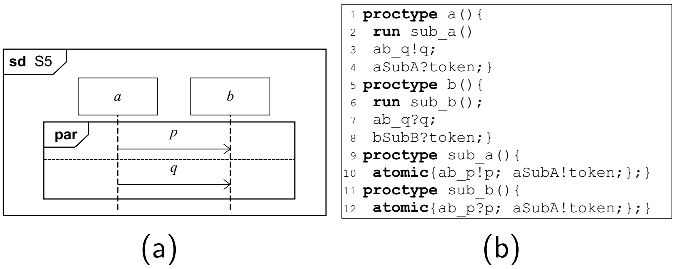
2.4 Parallel Combined Fragments

A Parallel Combined Fragment, denoted by **par** operator, represents a parallel merge between the behaviours of the operands. The events of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved [17]. Its set of traces describes all the ways that events of the operands may be interleaved without obstructing the order of the events within the operand [10]. Eq. 3 shows the set of possible traces of the diagram in Fig. 4(a).

$$\begin{aligned} S5 = \langle !p, ?p\rangle \ \mathbf{par} \ \langle !q, ?q\rangle = &\{\langle !p, ?p, !q, ?q\rangle, \langle !p, !q, ?p, ?q\rangle, \langle !q, !p, ?q, ?p\rangle, \\ &\langle !q, ?q, !p, ?p\rangle, \langle !q, !p, ?p, ?q\rangle, \langle !q, !p, ?q, ?p\rangle\} \end{aligned} \tag{3}$$

2.4.1 PROMELA Representation:

Parallel behaviour can be implemented using sub-instances of the lifelines covered by the parallel fragment (Fig. 4(b), lines 9 and 11). The new element is instantiated right before the main process starts the parallel activities. The actions inside the parallel fragment are divided among the main process and its sub-instances. Each one executes one operand. A synchronism mechanism should be implemented to ensure that no event after a combined fragment will overtake an event in it. This synchronism is done with token messages that will be sent from the subprocess to the main process right before finishing its execution (Fig. 4(b), lines 10 and 12). The main process must wait for all tokens before continuing the execution (Fig. 4(b), lines 4 and 8). Fig. 4(b) shows the PROMELA code of the model in Fig. 4(a).

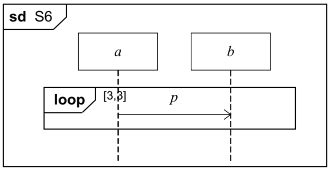


```
1 proctype a(){
2  run sub_a()
3  ab_q!q;
4  aSubA?token;}
5 proctype b(){
6  run sub_b();
7  ab_q?q;
8  bSubB?token;}
9 proctype sub_a(){
10  atomic{ab_p!p; aSubA!token;};}
11 proctype sub_b(){
12  atomic{ab_p?p; aSubA!token;};}
```

Fig. 4. (a) Parallel Combined Fragment, (b) Respective PROMELA Code

2.5 Loop Combined Fragments

The operator **loop** indicates that the combined fragment represents a repetition structure. The loop operand will be repeated a certain number of times according to the values defined by the designer. The loop construct represents a recursive application of the **seq** operator where the loop operand is sequenced after the result of earlier iterations [17].



```
proctype a(){
 byte i=0;
 do
  ::(i<N) -> atomic{ab_m!m;i++;};
  ::else -> break;
 od}
proctype b(){
 byte i=0;
 do
  ::(i<N) -> atomic{ab_m?m;i++;};
  ::else -> break;
 od}
```

(a) (b)

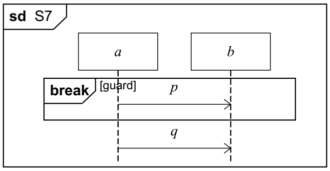
Fig. 5. (a) Loop Combined Fragment, (b) Respective PROMELA Code

2.5.1 PROMELA Representation:

Our PROMELA implementation of loop works with a fixed number of repetition. PROMELA defines *do* operator as a repetition construct. Loop fragments are implemented by declaring a global variable with the total number of repetition, and a *do* structure in each lifeline covered by the fragment. Fig 5(b) presents the PROMELA code of the model in Fig. 5(a).

2.6 Break Combined Fragments

The interaction operator **break** shows a combined fragment representing a breaking scenario. If the guard condition is true, the operand scenario is performed instead of the remainder of the enclosing interaction fragment [17].



```
proctype a(){
 if
  ::(guard) -> atomic{ab_p!p;
                      goto L0;};
  ::else -> skip;
 fi;
 ab_q!q;
 L0:skip;}
proctype b(){
 if
  ::(guard) -> atomic{ab_p?p;
                      goto L1;};
   ::else -> skip;
 fi;
 ab_q?q;
 L1:skip;}
```

(a) (b)

Fig. 6. (a) Break Combined Fragment, (b) Respective PROMELA Code

2.6.1 PROMELA Representation:

The **break** operator can be simulated with *goto* statements in PROMELA. If the guard condition is true, the action inside the break combined fragment is performed, then the execution jumps to the end (lines 4 and 12 of Fig.6(b)). The non-deterministic behavior is implemented in the same way as in alternative and option combined fragments. Fig. 6(b) shows the PROMELA code for the break combined fragment in Fig. 6(a).

3 Using Source/Destination and Send/Receive Events for Sequence Diagrams V&V

In the previous section, we provided what is needed to simulate the execution of sequence diagrams by covering the most important combined fragments. However, the main objective of using PROMELA-based model is not to simulate the execution of sequence diagrams, but the verification of formal properties. When it comes to verify formal properties on SPIN, it is impossible to determine whether a send or receive event has occurred. Indeed, the system state does not change when messages are sent over channels [23]. To overcome this obstacle, [23] proposed a flag-based technique to mark an occurrence of a send/receive event. This section presents an extension of this approach that is able to determine *who is sending/receiving what to/from whom* at any time of the execution. This information is very useful when one wants to write properties to be verified. We also show how to write LTL properties using this approach.

3.1 Tracking the execution state

The first step toward the formal V&V of sequence diagrams is to keep track of the actions performed by the entities in the interaction. In other words, it is essential to be aware of all event occurrences during the execution. In [23], the authors suggest tracking of sending and receiving events of messages by using flags associated with the respective event (e.g., using the flag "Sx" for "sending message x"). In spite of the fact that they improved the set of properties that can be verified, many other properties are still not covered since they require the information of the entities the are interacting (e.g., the following constraint could be specified to a particular system: "*Alice* is not supposed to receive a request from *Bob*"). In [2], the authors define the concatenation of sender, message and receiver as one action, but they do not include send and receive events. Even though it provides the entities information, it does not give the flexibility to write properties looking only at a particular element in the model. (e.g., *Server* does not send anything to anyone without signing). This flexibility is important because systems usually have many entities, but only some of them are really critical. To address these weaknesses, we define a state transition system such that the transitions are trigged by the send and receive events of the interaction and each state is characterized by a 4-tuple consisting of the following fields:

(i) Lifeline that performed the last action.

(ii) Last performed action (send or receive).

(iii) Message used in the last action.

(iv) Lifeline to/from which the message was sent/received.

Each state contains the information we need to track, and each field can be used separately.

In PROMELA, we represent each state as a set of flags. For each lifeline, each message, and *send/receive* events a flag is declared. The values of these flags are updated together with each send/receive event. The update is done using a *d_step* statement to make the assignment of all new values as one step at the execution time. Fig. 7 shows the PROMELA code of the interaction in Fig. 2(a) with its flags to represent states.

```
proctype a(){
 atomic{d_step{send=1; receive=0; msg_p=1; msg_q=0; proc1_a=1;
               proc1_b=0; proc2_a=0; proc2_b=1;}; ab_p!p;};
 atomic{d_step{send=1; receive=0; msg_p=0; msg_q=1; proc1_a=1;
               proc1_b=0; proc2_a=0; proc2_b=1;}; ab_q!q;};}
proctype b(){
 atomic{ab_p?p;d_step{send=0; receive=1; msg_p=1; msg_q=0;proc1_a=0;
                      proc1_b=1; proc2_a=1; proc2_b=0;};};
 atomic{ab_q?q;d_step{send=0; receive=1; msg_p=0; msg_q=1;proc1_a=0;
                      proc1_b=1; proc2_a=1; proc2_b=0;};};}
```

Fig. 7. PROMELA code of the diagram in Fig. 2(a)

3.2 Using flags to specify LTL properties

After defining a methodology to track the execution state, LTL formulas can be written in terms of boolean expressions over the flags. For example, if one wants to say "*b* sends *p* to *a*", he/she should write the following expression: (proc1_b∧send∧msg_p∧proc2_a). A very useful property of the flag-based state is the ease of expressing sentence over all lifelines, or all messages, or all actions only by omitting the respective element in the expression. For example, if one wants to verify if "no lifeline receives messages from *a*", the respective expression is: ¬(receive∧proc2_a). This example also shows that the proposed technique gives the flexibility to write properties looking at a particular element in the model (lifeline *a* in that case). Therefore, the proposed flag-based mechanism to track execution state not only provides developers with the information they need to write properties, but also it allows them to specify properties in a very flexible way. Below, we present two examples of LTL properties written using flags and their respective verifications using SPIN model checker.

- **Example 1**: Suppose one wants to verify in the sequence diagram from Fig. 2(a) whether "no lifeline will send message q until b receives message p". The LTL formula corresponding to this property is:

$$\neg(\text{send} \wedge \text{msg\_q})\ \mathbf{U}\ (\text{proc1\_b} \wedge \text{receive} \wedge \text{msg\_p}) \tag{4}$$

After model checking, SPIN reports that the property does not hold. The counterexample that is returned is shown in Fig. 8(a).

- **Example 2**: In the model shown in Fig. 4(a), suppose one needs to verify if "always, after b receives p, eventually b receives q from a". The LTL formula expressing this property is:

$$\Box((\text{proc1\_b}\wedge\text{receive}\wedge\text{msg\_p}) \rightarrow \Diamond(\text{proc1\_b}\wedge\text{receive}\wedge\text{msg\_q}\wedge\text{proc2\_a})) \tag{5}$$

After model checking, SPIN shows that this property does not hold. The counterexample is presented in Fig. 8(b).

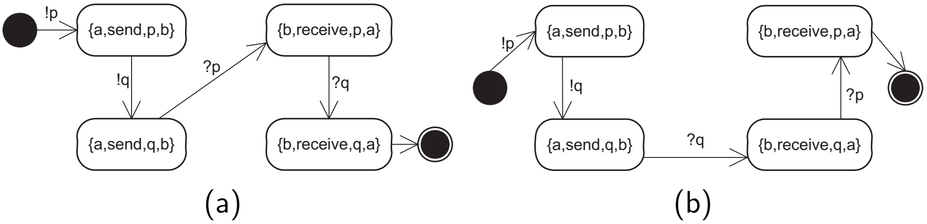


Fig. 8. SPIN counterexamples

4 Case Study

This section presents part of the system presented in [1]. In that thesis, the authors show the design of an Automated Teller Machine (ATM). The ATM interacts with two other entities: The Customer (User) and the bank. Fig. 9 describes a use case where the user starts a request by inserting his/her card. The ATM must verify the card and the personal identification number (PIN) to proceed. If the verification fails the card should be ejected. Otherwise, the user has the choice to perform some operations and the card is retained in the machine until the user finishes the transactions. The first and second combined fragments are dealing with the authentication of the card and the PIN, respectively. The third one shows an interaction using "cash in advance" operation.

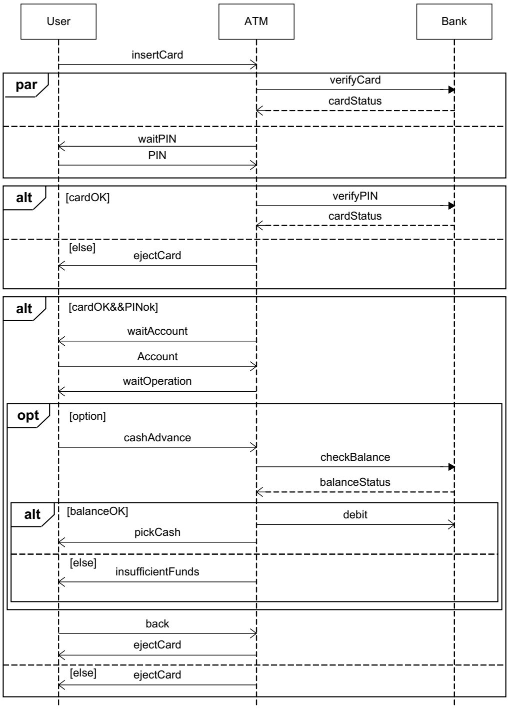


Fig. 9. ATM Sequence Diagram

4.1 LTL properties

(i) The first property states that the ATM cannot allow the user to request an operation if either the card or the PIN is not valid:

$$\Box(x \rightarrow \neg\Diamond y) \tag{6}$$

where $x = (\text{start} \wedge (\neg\text{cardOK} \vee \neg\text{PINok}))$, and $y = (\text{proc1\_user} \wedge \text{receive} \wedge \text{msg\_waitAccount})$.

(ii) The second property is needed to avoid inconsistencies between the money given to the user and the amount debited in the bank. It asserts that the ATM must first debit the amount in the bank, and then give the money to the user. In other words, the user does not receive pickCash until the bank receives debit:

$$\neg x \ \mathbf{U} \ y \tag{7}$$

where $x = (\text{proc1\_user} \wedge \text{receive} \wedge \text{msg\_pickCash})$, and $y = (\text{proc1\_bank} \wedge \text{receive} \wedge \text{msg\_debit})$.

(iii) The fourth property deals with the usability of the ATM. It states that, if the ATM receives insufficient funds, it should allow the user to choose other operation before finishing the session:

$$\Box(x \rightarrow (\neg y \ \mathbf{U} \ w)) \tag{8}$$

where x = (proc1_user ∧ receive ∧ msg_insufficientFunds), y = (end), and w = (proc1_atm ∧ send ∧ msg_waitOperation).

(iv) The third property is to ensure the correct end of the session between the ATM and the user. It says that, after the user receives ejectCard, the ATM cannot send anything to the user:

$$\Box(x \rightarrow \neg\Diamond y) \tag{9}$$

where x = (proc1_user ∧ receive ∧ msg_ejectCard), and y = (porc1_atm ∧ send ∧ proc2_user).

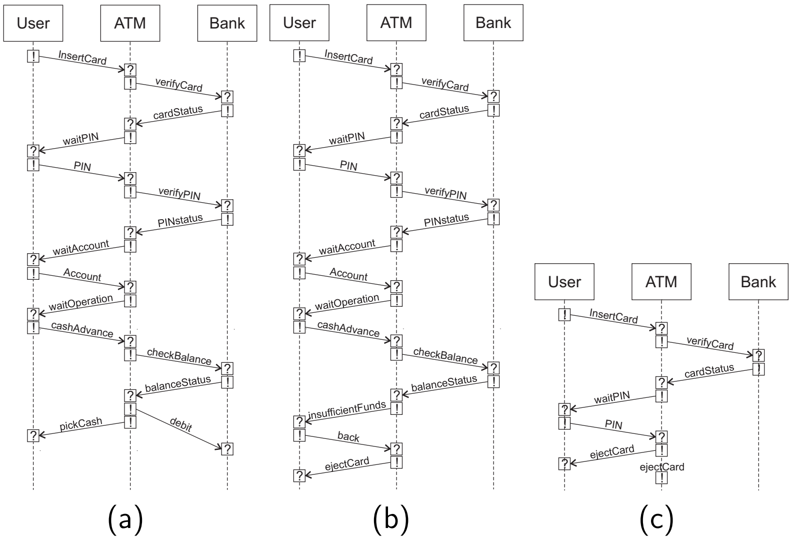


Fig. 10. SPIN counterexamples for LTL properties: (a)counterexample of property ii, (b) counterexample of property iii, (c) counterexample of property iv

4.2 ATM Case Study Results

Using SPIN to verify the properties described previously, we found that only the first property is satisfied. The model checker was able to provide a counterexample for each of the other properties. In the following, we present the failing trace related to the verification of each property.

- **Property ii Counterexample**: In the trace shown in Fig. 10(a), it possible to see that there is at least one execution path on which the user receives the money before the bank receives the message to debit. If, for some reason, the message to debit is not delivered, the ATM will be not able to ask the user to give the money back.

- **Property iii Counterexample**: The counterexample of Fig. 10(b) shows that, after receiving insufficient funds, the user does not have the opportunity to re-enter a different operation. It means that, if the user does a mistake, he/she needs to restart the whole operation from the beginning.
- **Property iv Counterexample**: Fig. 10(c) presents a counterexample where the ATM tries to eject the card twice. This inconsistent behavior should be eliminated from the design.

In order to illustrate the performance of our approach regarding this case study, Table 2 shows a summary of the results along with the number of states, number of transitions, memory and time used by SPIN to perform the verification. It is possible to see that the maximum time spent for verification was 0.132s, which is very reasonable. However, it is well-known that model checkers can have state explosion when verifying bigger models. There are some techniques to optimize the PROMELA specification to avoid state explosion, but those techniques will be considered in a future work.

Property	Result	Stored states	Matched states	Transitions	Memory Usage	Time
i	Passed	10991	11389	22380	3.673MB	0.13s
ii	Failed	88	4	92	2.501MB	0.06s
iii	Failed	986	1211	2197	2.598MB	0.072s
iv	Failed	7864	9344	17208	3.379MB	0.132s

Table 2
Summary of the results

5 Tool support

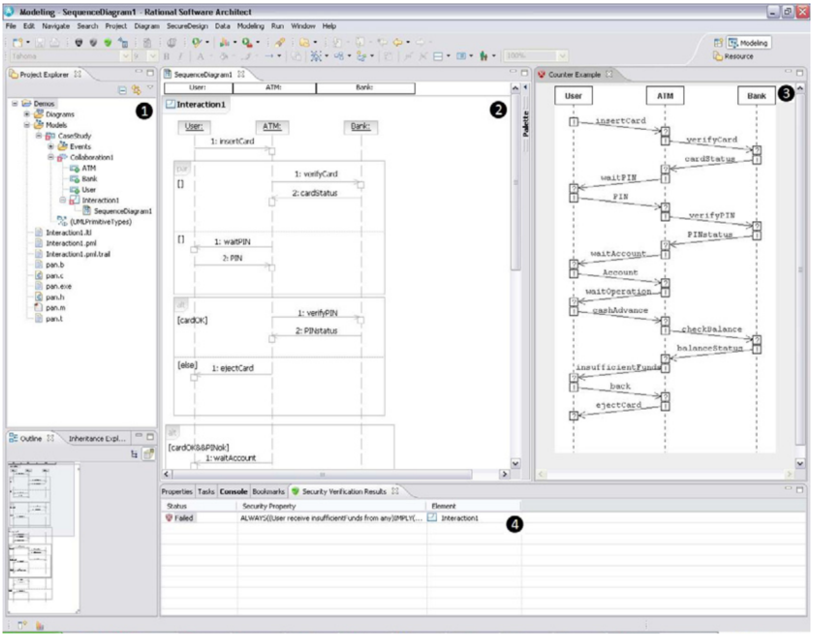
In this section, we provide details about the implementation of the tool to support the V&V of sequence diagrams. We have chosen IBM Rational Software Architect[4] (RSA) as the environment for development, since it contains a very powerful UML modeller. In addition, it can be augmented with Eclipse plugins, which allows the verification engine to be embedded into the development environment.

The tool is implemented as an Eclipse plugin composed of three main parts:

(i) A property editor to allow the developer to assign properties to UML diagrams.

(ii) A UML translator to generate the PROMELA code from UML interaction.

[4] http://www.ibm.com/software/awdtools/architect/swarchitect/



Status	Security Property	Element
Failed	ALWAYS((User receive insufficientFunds from any)IMPLY(...	Interaction1

Fig. 11. Screenshot of the IBM RSA workspace

(iii) A result interpreter to read the output of the SPIN model checker and translate it into an easy-to-understand graph to be analysed by the developer.

Fig. 11 shows a screenshot of the IBM RSA workspace. The labels 1 and 2 of this figure show the Project explorer and the UML diagram editor, respectively, provided by the IBM RSA. The labels 3 and 4 show contributions of our plugin to the workspace, where label 3 shows the summary of the verification result displaying which properties failed or passed, and label 4 shows the graph of the counterexample that is automatically generated by our tool.

It is important to mention that, for the developer point of view, there is no need to understand the mechanism that performs the verification. The translation to PROMELA, the execution SPIN and the interpretation of the SPIN's output are all done in background. The developer should only assign properties to diagram, start the verification and assess the results.

6 Related Work

In the literature there is a considerable number of research works intending to verify UML diagrams. In [2], a framework is proposed for V&V of some popular UML diagrams (Class, State Machine, Activity and Sequence diagrams). In this approach, a semantics model called configuration transition system (CTS) is extracted from behavioral diagrams and then translated into NuSMV [5] code. This approach allows V&V of behavioral models against properties written in computational tree logic (CTL). Even though this approach is dealing with some UML 2.0 sequence diagrams elements, the proposed semantics model is not in full accordance with the standard semantics specified in [17] due to the lack of send and receive events. As a consequence, some traces, like those shown in Eq. 1, Eq. 2 and Eq. 3, cannot be captured in this approach. In [15], a detailed description of the translation of Message Sequence Charts (MSCs) [11] into PROMELA is presented. Since the MSCs are the basis of UML sequence diagrams [18], many of the proposed translation decisions can be applied to sequence diagrams. However, the proposed approach deals only with the basic components and decisions; consequently, its PROMELA representation of MSCs does not cover the behaviour of combined fragments presented in the sections 2.4, 2.5 and 2.6. [22, 23] propose a set of techniques to improve the quality of sequence diagrams. One of these techniques is trace analysis by using model checkers. To get PROMELA code from sequence diagrams, this technique provides a translation scheme that is based on [15]. However, since this approach is intended to UML 1.5 diagrams, combined fragments are not taken into account. In addition, the authors do not propose a mechanism to use source and destination for writing formal properties.

In [6, 13, 19], they determine whether a given interaction can be successfully executed in a system where the behaviour is specified using state machines. These works assume sequence diagrams as properties to be verified, whereas our approach has sequence diagrams as the target of verification.

In [4, 8, 9] and [12, 14, 16, 20], the translation into PROMELA and V&V using SPIN is presented for activity diagram and state machine diagram, respectively. Those works might be used in the future when it is important to consider the connection among different diagrams in the verification process.

7 Conclusion and Future Work

We present in this paper an efficient technique for formal V&V of UML 2.0 sequence diagrams. Since it takes into account the most popular UML combined fragments, this approach allows the developer to detect flaw in more completed complex sequence diagrams. The mechanism introduced in this work to keep track of the execution state provides the information the developer needs to write LTL properties. Moreover, the way it was implemented gives flexibility to write very expressive properties. This technique, along with other V&V tools, can provide a very useful framework to detect errors at the design phase, resulting in software much more reliable in the end of the software development process.

As opportunities for future work, two main problems could be addressed. First, there is a substantial gap between natural language and properties written formal logics [3]. The use of graphical notations, as proposed in [3], macro language [2] and patterns [7] in properties specification should be considered as an important improvement on model checkers based tools to bring them to the industry level. Second, many security aspects are not usually encapsulated in only one type of diagram. Normally they are scattered in various diagrams of different kinds. An inter-diagram analysis might provide results that are much more consistent in the assessment of UML models.

References

[1] Alawneh, L., "Verification and validation in systems engineering : application to UML 2.0 activity and class diagrams," Master's thesis, Dept. of Electrical and Computer Engineering, Concordia University (2006).

[2] Alawneh, L., M. Debbabi, F. Hassane, Y. Jarraya and A. Soeanu, *A unified approach for verification and validation of systems and software engineering models*, in: *Proc. of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, 2006, p. 10pp.

[3] Autili, M., P. Inverardi and P. Pelliccione, *A scenario based notation for specifying temporal properties*, in: *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools* (2006), pp. 21–28.

[4] Cao, H., S. Ying and D. Du, *Towards model-based verification of bpel with model checking*, in: *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology* (2006), p. 190.

[5] Cimatti, A., E. M. Clarke, F. Giunchiglia and M. Roveri, *NUSMV: A new symbolic model verifier*, in: *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification* (1999), pp. 495–499.

[6] del Mar Gallardo, M., P. Merino and E. Pimentel, *Debugging UML designs with model checking*, Journal of Object Technology **1** (2002), pp. 101–117.

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, in: *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 411–420.

[8] Guelfi, N. and A. Mammar, *A formal approach for the verification of e-business processes with promela*, Technical Report TR-SE2C-04-10, Software Engineering Competence Center, University of Luxembourg, Luxembourg (2004).

[9] Guelfi, N. and A. Mammar, *A formal semantics of timed activity diagrams and its promela translation*, apsec **0** (2005), pp. 283–290.

[10] Haugen, Ø. and K. Stølen, *STAIRS - steps to analyze interactions with refinement semantics*, in: *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, LNCS **2863** (2003), pp. 388–402.

[11] International Telecommunication Union, *ITU Recomendation Z.120: Message Sequence Chart (MSC)* (2004).

[12] Jussila, T., J. Dubrovin, T. Junttila, T. Latvala, I. Porres and J. K. U. Linz, *Model checking dynamic and hierarchical UML state machines*, in: *Proceedings of MoDeV 2 a (2006)*, pp. 94–110.

[13] Knapp, A. and J. Wuttke, *Model checking of UML 2.0 interactions*, in: *Models in Software Engineering*, Springer Berlin / Heidelberg, 2007 pp. 42–51.

[14] Latella, D., I. Majzik and M. Massink, *Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker*, Formal Aspects of Computing **11** (1999), pp. 637–664.

[15] Leue, S. and P. B. Ladkin, *Implementing and verifying MSC specifications using promela/xspin*, in: *Proc. of of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*, 1997, pp. 65–89.

[16] Mikk, E., Y. Lakhnech, M. Siegel and G. J. Holzmann, *Implementing statecharts in promela/spin*, in: *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques* (1998), p. 90.

[17] Object Management Group, *UML 2.0 Superstructure Specification* (2007).

[18] Rumbaugh, J., I. Jacobson and G. Booch, "Unified Modeling Language Reference Manual, The (2nd Edition)," Pearson Higher Education, 2004.

[19] Schäfer, T., A. Knapp and S. Merz, *Model checking UML state machines and collaborations*, Electronic Notes in Theoretical Computer Science **55** (2001), pp. 357–369.

[20] Siveroni, I., A. Zisman and G. Spanoudakis, *Property specification and static verification of UML models*, ares **0** (2008), pp. 96–103.

[21] Stevenson, D. E., *Verification and validation of complex systems*, in: *Intelligent Engineering Systems through Artificial Neural Networks* (2002), pp. 159–164.

[22] van Amstel, M. F., "Design and Assessment of Analysis Techniques for UML Sequence Diagram," Master's thesis, Technische Universiteit Eindhoven (2006).

[23] Van Amstel, M. F., C. F. J. Lange and M. R. V. Chaudron, *Four automated approaches to analyze the quality of UML sequence diagrams*, in: *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2* (2007), pp. 415–424.