Automatic Generation of PROMELA code from Sequence Diagram with Imbricate Combined Fragments

Abstract : *Formal verification of UML diagram is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. The most widely used techniques for system or software verification: Simulation and testing, deductive verification and Model checking. Model checking is a formal verification technique, in which an abstract model of a system is testing automatically to verify whether this model meets a given specification. SPIN Model checker is a popular open-source software tool, used by thousands of people worldwide that can be used for the formal verification of distributed software systems, SPIN is understand PROMELA code and properties are express in Linear Temporal Logic(LTL). This article aims propose a method for converting UML sequence diagrams with imbricate combined fragment automatically to PROMELA code to simulate the execution and to verify properties written in Linear Temporal Logic (LTL) with SPIN Model checker.*

Key words: UML2.0, Sequence diagram, Imbricate combined fragment, Graph transformation, AToM3, PROMELA.

1. Introduction

One of the key issues in software development, like in all engineering problems, is to ensure that the product delivered meets its spécification. Verification and validation are well-established techniques for ensuring the quality of a product within the overall software development lifecycle.

Verification and validation (V&V) is concerned with answering two fundamental questions: did we build the right product, and did we build the product right?. Verification is a process that makes it sure that the software product is developed in the right way. The software should confirm to its predefined specifications. Validation is a process of finding out if the product being built is right? That is, whatever software product is being developed, it should do what the user expects it to do. The software product should functionally do what it is supposed to, it should satisfy all the functional requirements set by the user.

The Unified Modeling Language (UML) is a collection of semi-formal standard notations and concepts for modeling the software systems at different stages and views during their development. Beyond diagrams that represent the static structure of a system, it also defines diagrams to model the dynamic behavior of systems. In our approach, we focus to the Verification and Validation of one of the most popular UML diagrams: the sequence diagrams with imbricate Combined Fragments witch describe messages exchanged between objects to accomplish tasks. UML 2.0 adds many major structural controls construct to the Sequence Diagram, including Combined Fragments and Interaction Use, to allow multiple, complex scenarios to be aggregated in a single Sequence Diagram. Combined Fragments permit different types of control flow, such as interleaving and branching, for presenting complex and concurrent behaviors, increasing a Sequence Diagram’s expressiveness.

With models being expressed in the UML, the application of verification and validation is complicated because UML model is not an input language of a verification tool and not directly executable. Many techniques have been proposed for V&V of UML diagrams, for example static analysis, theorem proving and model checker. Model checking is an automated verification method to check whether a formally specified property holds for a model of a system. Another important contribution is the definition of the PROMELA (Protocol Meta Language) structure that provides a precise semantics of most of the newly UML 2.0 introduced combined fragments, allowing the execution of complex interactions. PROMELA is a high level language to specify a system description that is used by the software verification and simulation tool SPIN which designed for automatically verifying LTL formulas.

In our approach, graph transformation techniques are applied for automated translation of sequence diagrams with imbricate combined fragment to PROMELA model. Graph transformation is increasingly popular as a meta-language to specify and implement visual modeling techniques, such as the UML. We choose AToM3 (A Tool for Multi-formalism and Meta-Modeling) a tool implemented in Python to realize our idea which has a meta-modeling layer in which formalisms are modeled graphically and concrete syntax.

This article aims propose a method for converting UML sequence diagrams with imbricate combined fragment automatically to the SPIN model checking code PROMELA to simulate the execution and to verify properties written in Linear Temporal Logic (LTL).

2. Related Work

Several researchers have studied the Verification and Validation of UML diagrams [1, 2, 3] and in particular, an approach for the formal verification of UML diagrams, such as class, state machine and communication diagrams, is presented in [4]. In [5], a framework is proposed for V&V of some UML diagrams (Class diagram, State Machine, Activity diagram and Sequence diagram). A verification approach of the UML class and activity diagrams is illustrated for a simple protocol is introduced by B. Prasanta [6]. The activity diagrams are converted into an FSM (based on behaviors). Thereafter the FSM is converted into PROMELA through an intermediate language.

The most of the proposed approaches target only activity diagrams [7,8,9,10,11] and state machine diagrams [12,13,14,15,16,17,18]. There are some approaches targeting sequence diagram and message sequence charts. Such as M.F. van Amstel [19] which introduces an approach to improve the quality of UML sequence diagrams using SPIN and PROMELA for the verification. In [20], the state machine is converted into PROMELA code as a protocol model and its properties are derived from the sequence diagram as Linear Temporal Logic (LTL). Peter B.Ladkin and Stefan Leue [21], present a description of the translation of Message Sequence Charts (MSCS) into PROMELA. Since of MSCS is an interaction diagram from the SDL (Specification and Description Language) family very similar to UML's sequence diagram. Yet, the proposed approach trait only with the basic components but its PROMELA representation of MSCS does not cover the combined fragments. A formal verification technique for UML 2.0 sequence diagrams employing linear temporal logic (LTL) formulas and the SPIN model checker to reason about the occurrences of events is introduced by Lima et al. [22].

However, the proposed approach, present the trace semantics of the most popular combined fragments with imbrications and their respective PROMELA code that correctly simulates the execution traces using AToM3 graph transformation tool.

3. Proposed approach

Models drawn in a sequence diagram are automatically converted to PROMELA the input language of the SPIN model Checker using Linear Temporal Logic (LTL) to express the validity of models. Graph transformation techniques are applied to realize the converter using AToM3.

Figure 1 shows the flow of automatic conversion from UML sequence diagram with imbricate combined fragment into a PROMELA model.

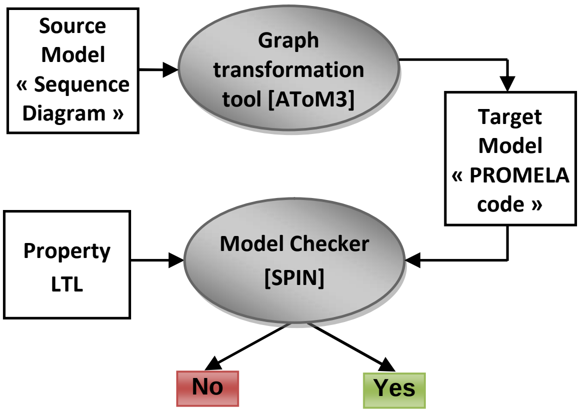

Source Model « Sequence Diagram »
Graph transformation tool [AToM3]
Target Model « PROMELA code »
Property LTL
Model Checker [SPIN]
No
Yes

Figure 1: Overview of our approach.

4. Graph transformation with AToM3 tool

Graph transformations is the approach that emerges from a natural and intuitive way among the model transformation approaches, this is due to the nature of the two concepts. The graph transformation is a process of graph rewriting based on graph grammars. A graph grammar is simply a result of well-formed rule, by analogy to Chomsky grammars where words are replaced by graphs and term rewriting is replaced by the bonding graph. Graph grammars are composed of production rules each having graphs in their left and right hand sides (LHS and RHS). The host graph is an input graph which compared with the rules. A rewriting system iteratively applies matching rules in the grammar to the host graph and replaces the sub graph by the RHS until no more rules are applicable.

AToM3 is a Meta-Modeling tool. As it has been implemented in Python, it is able to run (without any change) on all platforms for which an interpreter for Python is available: Linux, Windows and Mac OS. The main idea of the tool is: “everything is a model”. During its implementation, the AToM3 kernel has been bootstrapped from a small initial kernel. Models were defined for boots trapped parts of it; code was generated and then later incorporated into it. Also, for AToM3 users, it is possible to modify some of these model defined components, such as the meta-formalisms and the user interface. The main component of AToM3 is the Kernel, responsible for loading, saving, creating and manipulating models (at any meta-level, with the Graph Rewriting Processor and graph grammar models), as well as for generating code for customized tools. This code (meta-models and meta-meta-models) can be loaded into AToM3.

5. UML sequence diagram

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Table 1 describes the elements that you can see on a sequence diagram with combined fragment (CF).

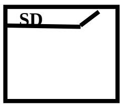
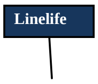
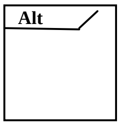
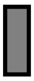
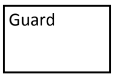
Element	**Description**	**Represen-tation**
Interaction	The collection of messages and lifelines that is displayed in the sequence diagram.	
Line Life	A lifeline represents an individual participant in the Interaction.	
Combined Fragment	A combined fragment is used to group sets of Messages together to show conditional flow in a Sequence diagram.	
Message	A message defines a particular communication between Lifelines of an Interaction.	
Execution Specification	A participant that is external to the system that you are developing.	
Operand	Defines the content of a combined fragment.	

Table 1: Basic element of sequence diagram

UML 2.0 has introduced significant improvements to the capabilities of sequence diagrams. Most of these improvements are based on the idea of interaction fragments which represent smaller pieces of an enclosing interaction. Multiple interaction fragments are combined to create a variety of combined fragments, which are then used to model interactions that include parallelism, conditional branches and optional interactions. Combined Fragments permit different types of control flow, such as interleaving and branching, for presenting complex and concurrent behaviors, increasing a Sequence Diagram’s expressiveness. Through the use of Combined Fragments the user will be able to describe a number of traces in a compact and concise manner. The execution of Occurrence Specifications (OS) enclosed in a CF is determined by its Interaction Operator, which is summarized as follows [23]:

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

In our approach we focus to some combined fragment such as **Alternative, Weak sequencing, Loop, Option and Break.**

6. PROMELA Representation

The PROcess MEta LAnguage (PROMELA) is a high level language to specify system descriptions that is used by the software verification and simulation tool SPIN. We choose PROMELA/SPIN because PROMELA provides all necessary concepts (sending and receiving primitives, parallel and asynchronous composition of concurrent processes and communication channels) that were necessary to implement the sequence diagram with combined fragment [21].

Table 2 provides the representation of the basic elements of the sequence diagram and their combined fragment in PROMELA. The conversion presented here is an adaptation of the one presented in [21].

UML element	PROMELA element	PROMELA statement
Lifeline	Process	proctype {...}
Message	Message	mtype ={m1,...,mn}
Connector	Communication channel for each message arrow	chan chan1 = [1] of {mtype}

Send and receive events	Send and receive operations	Send-> ab_msg1!Msg1;
		Receive-> ab_msg1?Msg1;

Table 2: Mapping of basic UML sequence diagrams into PROMELA [21]

7. Meta-model sequence diagram

The meta-models in AToM3 are a UML class diagrams and the constraints are expressed in Python language. We proposed the meta-model sequence diagrams containing five classes such as **interaction** is a global model containing the remaining elements and it is represented by a set of **lifelines**, **executionspecification** which refers to the period of activity, **combinedfragement** spans over many lifelines and it has one or more operands. A combined fragment with operator *option*, *loop or neg* contains exactly one operand, while for other operators it contains an arbitrary number of operands, each **operand** has a guard attribute and spans over a subset of the lifelines which it's combined fragment spans over. The relation **CFContain, OpContain** and **IContain** allow the combination or overlapping fragments combined so to define relationships (father / child), **Connect** represent the relation between periods of activity and the lifeline or between two periods of activity, **message** consists of a send event and a receive event, which are normally placed on two different lifelines as show as in Figure 3. Figure 2 shows our simplified meta-model for UML 2.0 sequence diagram.

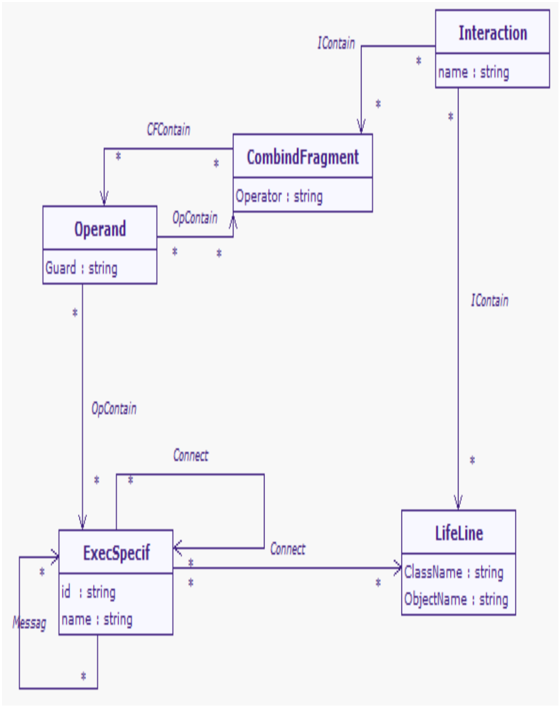


Figure 2: A simplified meta-model for UML 2.0 sequence diagrams

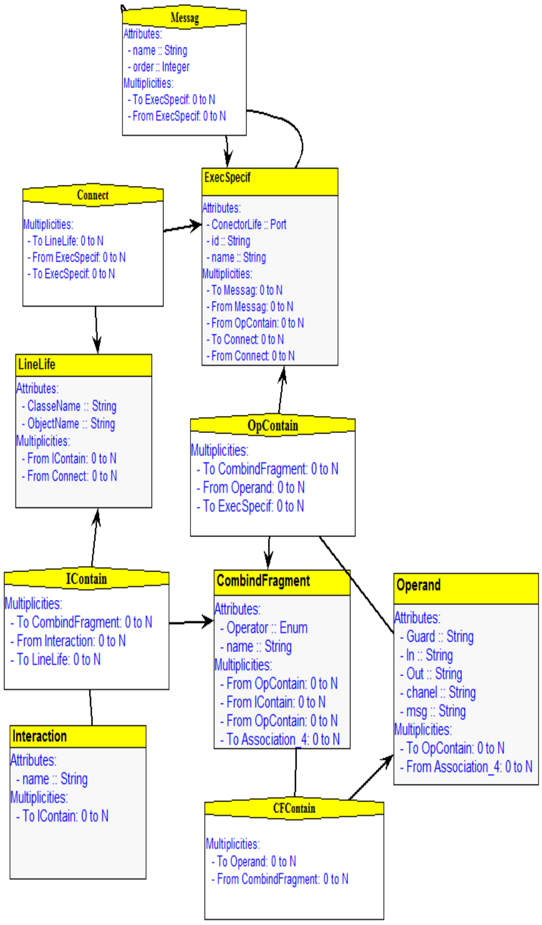


Figure 3: Meta Model of sequence diagram with AToM3

8. Transformation rules

In this section we present the rules of transformation and we show how the rules gradually transform from a sequence diagram and the imbrications of FC into PROMELA code.

8.1. Create file rule

This rule permits the creation of file for each proctype. The Listing 1 shows the action of this rule.

```
node = self.getMatched(graphID,
self.LHS.nodeWithLabel(1))
node.visited=1
cgd = self.graphRewritingSystem.parent.codeGenDir
self.graphRewritingSystem.file
=open(cgd+"/proctype"+node.ObjectName.toString()
+".txt","w+t")
file = self.graphRewritingSystem.file
file.close()
```

Listing 1

8.2. Messages and channels declaration rule

Figure 4 represent the input model of the rule transformation for declaration of messages and channels in PROMELA. For lack of space we only describe in the following some rules.

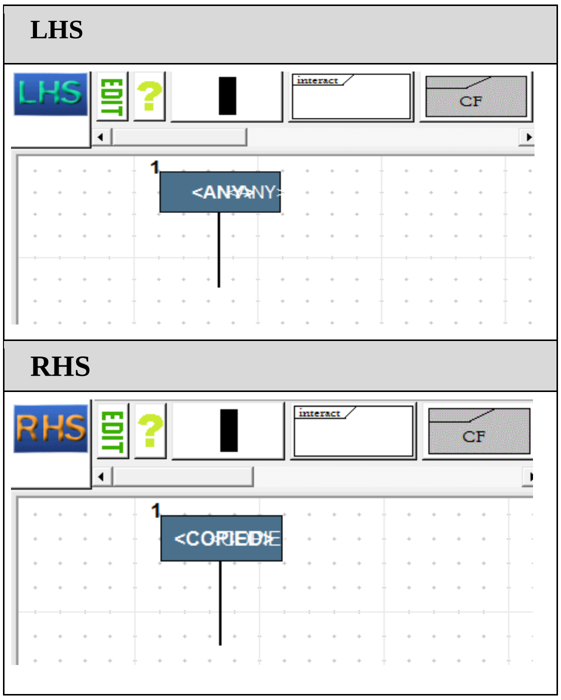
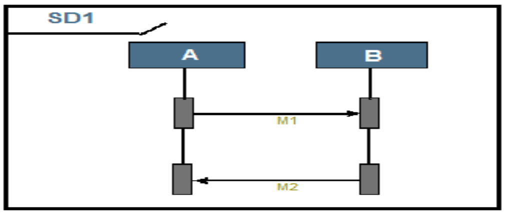


Figure 4: Translation rule for message and channels to PROMELA code.

Figure 5 represents a simple example and their respective PROMELA code after the application of this rule.

Source



Target

```
/*Message declaration */
mtype={ M1 ,M2 };
/* Channels Declaration*/
chan AB_M1 =[1] of {mtype};
chan BA_M2 =[1] of {mtype};
```

Figure 5: simple example

8.3. Combined Fragment rule

The goal of this rule is converting the combined fragment with imbrications to PROMELA, we refer to [22] for shows each combined fragment and their respective PROMELA code. To keep the execution traces and the conditional flow of imbricate combined fragment we use the following methods ***getHierChildren()*** and ***getAllHierChildren().***

Figure 6 represents the input model of the rule transformation for combined fragment.

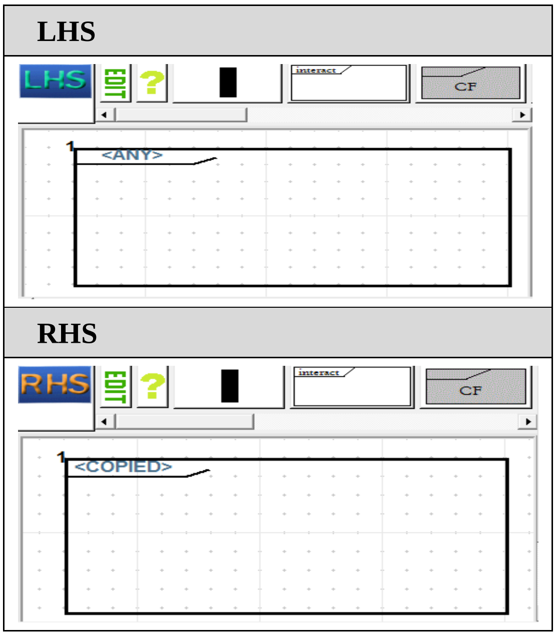


Figure 6: Translation rule for deferent combined fragment to PROMELA code.

The conversion is illustrated by an algorithm represented in Listing 2. We define a number of methods to realize this rule such as method **grand()** to extract the grand parent of all diagram's elements as shows in Listing 3, and method **boite()** to extract the message related with the ExecSpecif as shows in Listing 4. Due to space constraint the Python code corresponding to the action of this rule cannot be represented in this paper.

```
We suppose :
    Cff: list of combined fragment.
    Nop: list of ExecSpecif.
    Msgl: list of message.
    Inter: list of interaction.
    grand (): method to extract the grandparent.
    Boite(): method to extract all messages related
             with ExecSpecif.
begin
 for I in grand() do:
    If I in Cff then:
      G=i.type()
      If g="alt" then:
        Alt()
       Else if g="Seq" then:
        seq()
      Else if g="option" then:
        option()
      Else if g="break" then:
        break ()
      Else if g="loop" then:
        loop()
    else :
      msg=Boite(i)
      write the corresponding PROMELA code of
      msg in the file.
end
```

Listing 2

```
#definition of method to extract the grand Parent.
def grand(self):
  lst = self.getHierParent()
  if lst != None and lst.getHierParent() not in Inter :
     return Tout(lst)
  else:
   return lst
```

Listing 3

```
def boite(boite):
 t=""
 ll=[]
 for b in Msgl:
      if b.in_connections_[0] not in ll and
b.out_connections_[0] not in ll :
        ll.append(b.in_connections_[0])
        ll.append(b.out_connections_[0])
        if boite in ll:
          t=b
          break
 return t
```

Listing 4

9. Example

We will illustrate our approach at the hand of a simple UML model shown in Figure 7. After the application of the previous grammar we have obtained the PROMELA code as indicated by Listing 5.

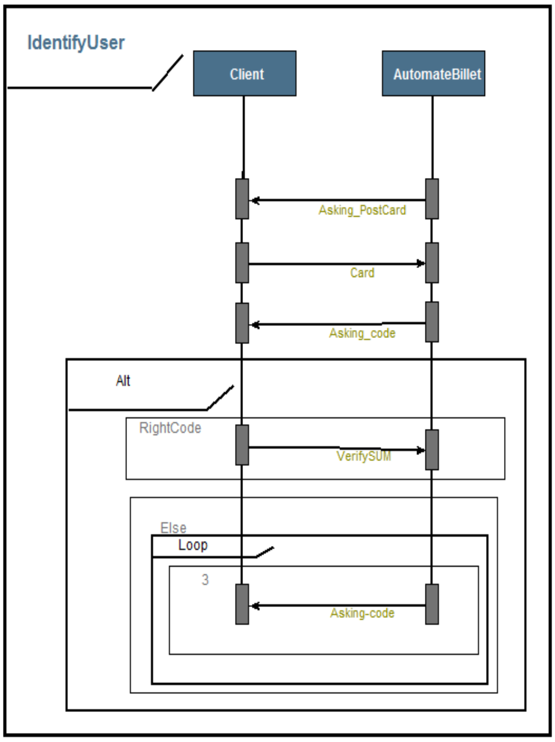


Figure 7: Example of sequence diagram.

```
/*Message declaration */
 mtype={ Asking_PostCard ,Card ,Asking_code,VerifySUM
};

/* Channels Declaration*/
chan AutomateBilletClient_Asking_PostCard =[1] of
{mtype};
chan ClientAutomateBillet_Card =[1] of {mtype};
chan AutomateBilletClient_Asking_code =[1] of {mtype};
chan ClientAutomateBillet_VerifySUM =[1] of {mtype};

/*LifeLines Specification */
proctype Client () {
AutomateBilletClient_Asking_PostCard?Asking_PostCard ;
ClientAutomateBillet_Card!Card ;
AutomateBilletClient_Asking_code?Asking_code ;
if
  ::(RightCode) ->
    ClientAutomateBillet_VerifySUM!VerifySUM ;
 ::else ->
      byte i=0;
      do
        ::(i<3) -> atomic {
           AutomateBilletClient_Asking_code?Asking_code ;
            i++;};
        ::else -> break;
      od
fi;
}
proctype AutomateBillet () {
AutomateBilletClient_Asking_PostCard!Asking_PostCard ;
ClientAutomateBillet_Card?Card ;
AutomateBilletClient_Asking_code!Asking_code ;
if
  ::(RightCode) ->
  ClientAutomateBillet_VerifySUM?VerifySUM ;
  ::else ->
     byte i=0;
     do
       ::(i<3) -> atomic {
           AutomateBilletClient_Asking-code!Asking-code ;
           i++;};
       ::else -> break;
     od
fi;
}
```

Listing 5

10. Conclusion

In this paper, we have suggested a method for automatic conversion of a model drawn in UML 2.0 sequence diagrams with imbricate Combined Fragments to PROMELA code. UML 2.0 adds many major structural controls construct to the Sequence Diagram, including Combined Fragments and Interaction Use, to allow multiple, complex scenarios to be aggregated in a single Sequence Diagram. Combined Fragments permit different types of control flow, such as interleaving and branching, for presenting complex and concurrent behaviors.

The transformation rules are used here to transform the host graph into final one by adding or removing vertices as specified in the predefined rules. We choose the tool AToM3 to run our case studies. We have shown how concrete syntax-based graph transformation rules can be used to specify a complicate transformation (the imbrication) implemented in the software tool AToM3

Near future work in UML sequence diagram is to add more transformation rules to manipulate the graph model of UML sequence diagram into other model.

Reference

[1] Igor Siveroni, Andrea Zisman and George Spanoudakis, “Property Specification and Static Verification of UML Models”, ARES '08 Proceedings of the 2008 Third International Conference on Availability, Reliability and Security.

[2] I. Paltor and J. Lilius. Vuml: A tool for verifying UML models. In ASE’99. IEEE, 1999.

[3] S. Wuwei, C. Kevinon, and H. James, “A toolset for supporting UML static and dynamic model checking”, 26th Annual International Computer Software and Applications Conference, 2002.

[4] M. Farid, G. Patrice, and B. Mourad, “Verifying UML diagrams with model checking: A rewriting logic based approach,” Seventh International Conference on Quality Software (QSIC 2007), pp. 356–362, 2007.

[5] L. Alawneh, 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, pp. 10.

[6] B. Prasanta, “Automated translation of UML models of architectures for verification and simulation using SPIN” 14th IEEE International Conference on Automated Software Engineering (ASE’99), pp. 102–109, 1999.

[7] N. Guelfi 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).

[8] Yutaka Yamada, Katsumi Wasaki. Automatic Generation of SPIN Model Checking Code from UML Activity Diagrams, 2011.

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

[10] H. Cao, 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), pp.190.

[11] I. Siveroni, A. Zisman and G. Spanoudakis, Property specification and static verification of UML models, 2008, pp. 96-103.

[12] Erich Mikk, Yassine Lakhnech, Michael Siegel. Implementing Statecharts in PROMELA/SPIN.

[13] Timm Schafer and Alexander Knapp and Stephan Merz. Model Checking UML State Machines and Collaborations.ENTCS, 55(3):357–369, 2001.

[14] D. Latella, 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] T. Jussila, 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.

[16] S. W. Vitus and J. Padget, "Symbolic Model Checking of UML Statechart Diagrams with an Integrated Approach," 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), pp. 337–346, 2004.

[17] Lilius, J. And I. P. Paltor, Formalising UML State Machines for Model Checking, in:R. B. France and B. Rumpe, editors, Proc. 2nd Int. Conf. UML, Lect. Notes Comp. Sci.1723 (1999), pp. 430–445.

[18] Kwon, G., Rewrite Rules and Operational Semantics for Model Checking UML Statecharts, in: A. Evans, S. Kent and B. Selic, editors, Proc. 3nd Int. Conf. UML, Lect.Notes Comp. Sci. 1939 (2000), pp. 528–540.

[19] M.F. van Amstel. Design and Assessment of Analysis Techniques for UML Sequence Diagrams.

[20] Prabhu Shankar Kaliappan, Hartmut Koenig. Designing and Verifying Communication Protocols Using Model Driven Architecture and SPIN Model Checker, 2008.

[21] Peter B.Ladkin and Stefan Leue, Implementing and verifying Message Sequence Charts 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.

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

[23] Hui Shen, Mark Robinson and Jianwei Niu. Formal Analysis of Scen nalysis of Scenario Aggregation.