From UML 2.0 Sequence Diagrams to PROMELA code by Graph Transformation Using AToM^3

Mouna Ait_Oubelli, Nadia Younsi, Abdelkrim Amirat, and Ahcene Menasria
University Centre Mohamed Chérif Messaadia, Souk-Ahras, Algeria
{aitoubelli.mouna@hotmail.com, nadia_younsi@hotmail.fr, amirat_karim@yahoo.fr, ahcene_menasria@yahoo.fr}

Abstract

A main challenge in software development process is to bring error detection to first phases of the software life cycle. The Verification and Validation (V&V) of UML diagrams is of interest in a number of applications such as detecting flaws at the design phase for software security, where it is crucial to detect security flaws before they can be exploited. In this paper, we propose an approach using a transformation tool to create a PROMELA code based model from UML interactions expressed in sequence diagram to use in SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL). Graph transformation is used as an approach of model transformation to propose a graph grammar for the translation using AToM3 tool.

Key words

UML2.0, Sequence diagram, Graph transformation, AToM3, PROMELA.

1. Introduction

If the software error is detected at the design phase before of the implementation, the software quality will acceptably be increased. For this target, the Verification and Validation (V&V) of UML [1] diagrams play a very important role in detecting flaws at the design phase. Verification is the process of determining that a simulation implementation accurately represents the developer's conceptual description and specifications. Verification answers the question: "*Are we developing the simulation right to our specifications?*". Validation is the process of determining the degree to which the relevant aspects of a simulation effectively represent the real world from the perspective of the intended uses of the simulation. Validation answers the question: "*Are we developing the 'right' simulation for the purposes we have determined?*".

The Unified Modeling Language is one of the well know and widely used standard describing software modeling in general and communication behavior in particular. The UML is a complete language that is used to design, visualize, construct and document systems. It is largely based on the object-oriented paradigm and is an essential tool for developing robust and maintainable software systems. The extended vision UML 2.0 introduces significant changes to interactions. In this study, we focus on formal V&V of one type of UML diagrams: "*sequence diagrams*".

The Sequence diagram describes messages exchanged between objects to accomplish tasks. Many techniques have been proposed for V&V of UML diagrams, for example static analysis, theorem proving and model checker. The Model checker is the most successful approach that's emerged for verifying requirements. 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 verification modeling language that allows the dynamic creation of concurrent process to model, for example, distributed systems. PROMELA models can be analyzed with SPIN model checker, to verify that the modeled system produces the desired behavior. We use graph transformation to approach the problem of transforming from sequence diagrams to PROMELA model. Graph transformation is increasingly popular as a meta-language to specify and implement visual modeling techniques, such as the UML. In this work, we use AToM^3 [2] (*A Tool for Multi-formalism and Meta-Modeling*) a tool which implements the idea presented above. AToM^3 has a meta-modeling layer in which formalisms are modeled graphically and concrete syntax.

This article aims to propose an approach that allows the generation a model code PROMELA for UML interactions expressed in a sequence diagrams using SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL).

2. Related work

Various works intending to verify UML diagrams. In [3], a framework is proposed for V&V of some UML diagrams (Class diagram, State Machine, Activity diagram and Sequence diagram). In [10], 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.

In [15], a metamodel-based transformation framework is proposed to implement the mapping from UML activity diagram to PROMELA. In [4, 5], the translation into PROMELA and V&V using SPIN is presented for activity diagram and in [6, 7, 8], the translation into PROMELA code from state machine diagram. Hermann [9] uses algebraic graph transformation, restricted to abstract syntax, to specify transformation rules for sequence diagrams. In [13], the translation into PROMELA from sequence diagrams by plug in Eclipse tool. In [14], the translation From UML 2 Sequence Diagrams to State Machines by Graph Transformation with AGG tool.

However, the proposed approach, present the trace semantics of the most popular combined fragments and their respective PROMELA code that correctly simulates the execution traces we use a Graph transformation tool AToM^3.

3. 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.

AToM^3 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 AToM^3 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 AToM^3 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 AToM^3 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 AToM^3.

4. Translation of UML 2.0 combined fragments into PROMELA

In this section, we present the trace semantics of some combined fragments and their respective PROMELA code that correctly simulates the execution traces as illustrated in Figure 1.

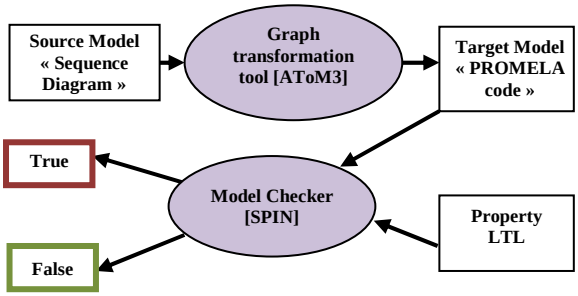

Figure 1. Overview of our approach.

4.1. Basic elements

Many techniques have been proposed for V&V of UML diagrams. The work presented in [10] specifies how to translate basic elements of MSCs into PROMELA [4] and [3] 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.

Figure 2 provides the representation of the basic elements of a sequence diagram with combined fragment (CF) which are translate to PROMELA with the key words: *proctype*: for declaring new process behavior, *mtype*: it defines symbolic names of numeric constants that are used as messages in the communicating process, *chan*: it declares and initializes communication channels. Finally, symbols **!/?** *Operators*: for sending/receiving messages to/from channels, respectively as shown in Table 1.

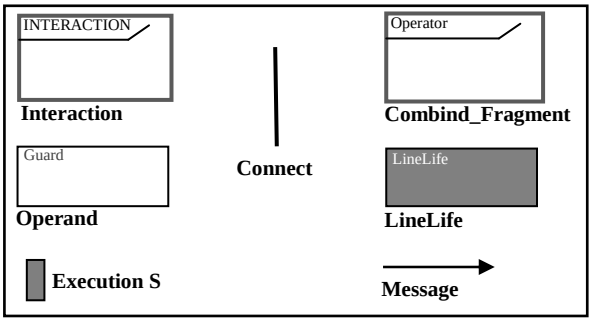

Figure 2. Elements of sequence diagram.

UML element	PROMELA element	PROMELA statement
Lifeline	Process	Proctype {....}
Message	Message	mtype ={m1,..,mn}
Connector	Communication channel for each message arrow	Chan chanName = [1] of {mtype}
Send and Receive events	Send and Receive operations	Send-> ab_msg1!msg1, Receive-> ab_msg1?msg1

Table 1.Mapping of basic UML sequence diagrams into PROMELA [14]

4.2. Meta-model sequence diagram

The meta-models in ATOM^3 are a UML class diagrams and the constraints are expressed in Python language. W proposed the meta model sequence diagrams containing five classes such as class interaction is a global model containing the remaining elements, class *Linelife*: define the line of life itself as participating in individual interaction, class *ExecutionSpecification* which refers to the period of activity, class *CombinedFragement* this is where we introduce the important and distinctive interactions in UML 2.0, *Operand* this class defines the content of a combined fragment, the relation *CFContain* and *IOContain* allow the combination or overlapping fragments combined so to define relationships (father / child), *Connect* represent the relation between periods of activity and the life line or between two active period as shown in Figure 3.

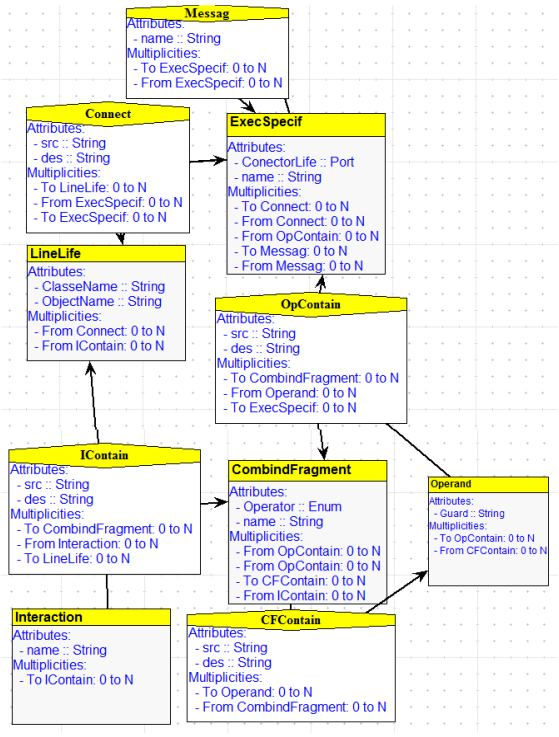

Figure 3. Sequence diagram meta-model.

5. Transformation rules

In this section we present the transformation rules, and we show how the rules gradually transform from a sequence diagram into PROMELA code. We use the example of Figure 5 to demonstrate our approach.

5.1. Messages and channels declaration rule

Figure 4 represent the input model of the transformation rule for declaration of messages and channels in PROMELA. We use the keyword **"mtype"** for messages and **"chan"** for channels.

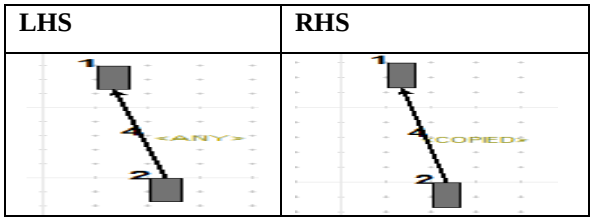

Figure 4. Translation rule for message and channels to PROMELA code.

For the example of Figure 5 we get PROMELA code indicated by Listing 1.

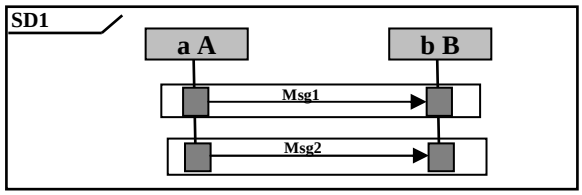

Figure 5. Simple interaction fragment.

```
mtype={Msg1,Msg2}
chan ab_Msg1=[1] of {mtype}
chan ab_Msg1=[1] of {mtype}
```

Listing 1

5.2. Linelifes specification

To represent the linelifes we use the keyword **"Proctype"** and the **!/? Operator** for sending/receiving messages to/from channels, respectively. For the example of Figure 5 we get PROMELA code indicated by Listing 2.

```
Proctype a (){ab_Msg1!Msg1; ab_Msg2!Msg2;};
Proctype b (){ab_Msg1?Msg1; ab_Msg2?Msg2;};
```

Listing 2

5.3. Translation combined fragments rules

A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram. In other words, it is a piece of an interaction [11]. Figure 6 represent the input model

of the transformation rule for deferent combined fragment to PROMELA code.

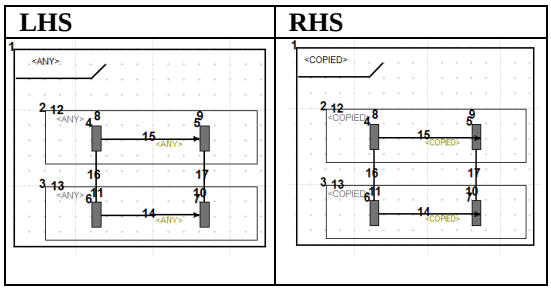

Figure 6. Translation rule for deferent combined fragment to PROMELA code.

5.3.1. Weak sequencing combined fragments

In Figure 7 we provide the execution traces for the weak sequencing operator which denoted by **Seq** operator. On a lifeline, the occurrence specification within an operand cannot execute until the OSs in the previous operand complete.

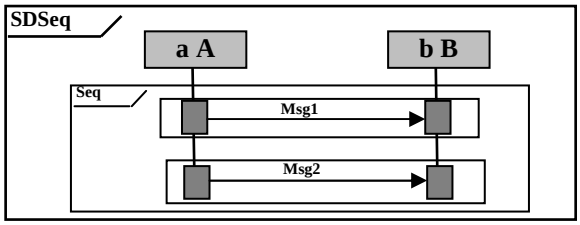

Figure 7. Weak sequencing combined fragment.

For the example of Figure 7 we get PROMELA code indicated by Listing 3.

```
Proctype a(){  ab_Msg1!Msg1; ab_Msg2!Msg2; }
Proctype b(){    ab_Msg1?Msg1; ab_Msg2?Msg2; }
init {atomic {run a();run b(); }}
```

Listing 3

5.3.2. Alternative combined fragments

Alternative combined fragments denoted by **Alt** operator, it represent a choice of behavior in sequence diagrams. It is one of the operands whose interaction constraints evaluate to true is nondeterministically chosen to execute. Each operand must have an explicit or an implicit or an else constraint. The chosen operand's constraint must evaluate to true. An implicit constraint always evaluates to true. The else constraint is the negation of the disjunction of all other constraints in the enclosing alternative combined fragment. The set of traces that defines a choice is the union of the traces of the operands [3,10].For the example of Figure 8 we get PROMELA code indicated by Listing 4.

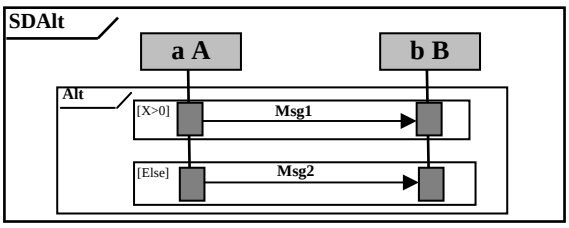

Figure 8. Alternative combined fragment.

```
Prototype a() {
 if
   ::(X>0) -> ab_msg2!msg2;
   :: else -> ab_msg1!msg1;
 fi;}
 prototype b() {
 if
   ::(X>0) -> ab_msg2?msg2;
   :: else -> ab_msg1?msg1;
 fi;}
 init {
   if
     :: (true) -> X>0=true;
     :: (true) -> X>0=false;
 fi;}
```

Listing 4

5.3.3. Parallel combined fragments

A parallel combined fragment, denoted by **Par** operator. The operand specifies on a Lifeline within different operands may be interleaved, but the ordering imposed by each operand must be maintained separately. 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. For the example of Figure 9 we get PROMELA code indicated by Listing 5.

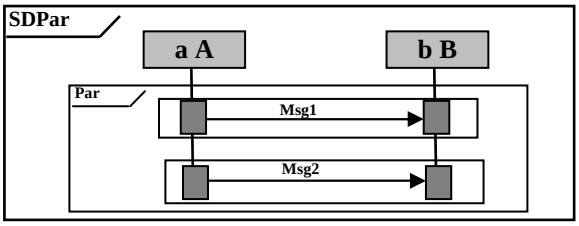

Figure 9. Parallel combined fragment.

```
Proctype a(){
 run sub_a()
 ab_Msg2!Msg2;
 aSubA?token;}
proctype b(){
 run sub_b()
 ab_Msg2!Msg2;
 bSuB?token;}
proctype sub_a(){
 atomic{ab_Msg1!Msg1; aSubA!token;};}
proctype sub_b(){
  atomic{ab_Msg1!Msg1; aSubA!token;};}
```

Listing 5

6. Example

We have applied our approach on the sequence diagram of Figure 10.

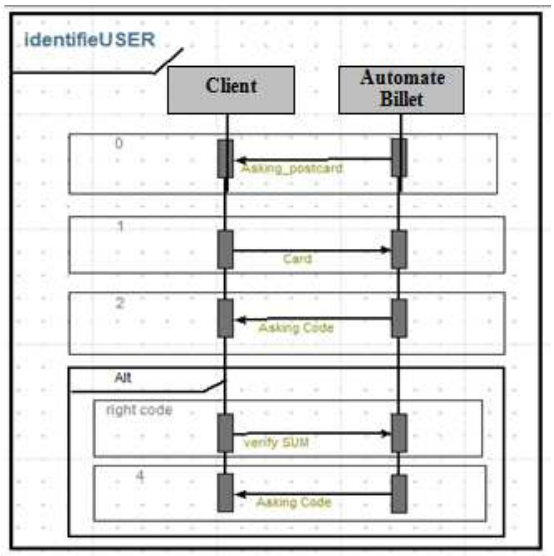

Figure 10. Example of sequence diagram.

6.1. Graph grammar for the transformation of UML sequence diagrams to PROMELA code

We have proposed a graph grammar containing 13 rules. To generate PROMELA code from a UML sequence diagram. For lack of space we only describe in the following some rules.

6.1.1. Messages declaration rule

Listing 6 represent the code python of Messages declaration rule's condition which represented in figure 4.

```
tnode=self.getMatched(graphID,
self.LHS.nodeWithLabel(4))
return tnode.visited == 0
```

Listing 6

And the action of this rule is indicated by Listing 7.

```
msg=self.getMatched(graphID,self.LHS.nodeWithLabel(4)
)
msg.visited = 1
posx,posy=
10+125*(self.graphRewritingSystem.NButtons%3),
10+70*(self.graphRewritingSystem.NButtons/3)
self.graphRewritingSystem.NButtons =
self.graphRewritingSystem.NButtons + 1
file = self.graphRewritingSystem.file
nameMSg=msg.name.toString()
file.write(nameMSg+" ")
```

Listing 7

6.1.2. Channels declaration rule

This rule used to declare channels which will be marked as "Visited" for the first time. We use the same condition and the graph grammar of messages declaration rule and the action indicated by Listing 8.

```
msg=self.getMatched(graphID,
self.LHS.nodeWithLabel(3))
l1=self.getMatched(graphID,
self.LHS.nodeWithLabel(1))
l2=self.getMatched(graphID,
self.LHS.nodeWithLabel(2))
msg.visited = 2
posx,posy=
10+125*(self.graphRewritingSystem.NButtons%3),
10+70*(self.graphRewritingSystem.NButtons/3)
self.graphRewritingSystem.NButtons=
self.graphRewritingSystem.NButtons + 1
file = self.graphRewritingSystem.file
nameMSg=msg.name.toString()
file.write("chan
"+l1.name.toString()+l2.name.toString()+"_"+nameMSg
+" =[1] of {mytype};\n")
```

Listing 8

6.1.3. Alternative combined fragments rule

Figure 6 represent the rule that translate the combined fragment **Alt** which are represented as if condition in PROMELA. The condition of the alternative combined fragments rules is indicated by Listing 9.

```
tnode=self.getMatched(graphID,
self.LHS.nodeWithLabel(1))
return tnode.visited == 0
```

Listing 9

Due to space constraint the Python code corresponding to the action of the alternative combined fragments rules cannot be represented in this paper.

6.2. PROMELA code result

After the application of the previous grammar we have obtained the PROMELA code as indicated by Listing 11 of the example represented in Figure 10.

```
mytype={ Asking_postcard ,Asking Code, Asking Code, Card verify SUM };
chan AutomateBilletClient_Asking_postcard =[1] of {mytype};
chan ClientAutomateBillet_Card =[1] of {mytype};
chan ClientAutomateBillet_verify SUM=[1] of {mytype};
chan AutomateBilletClient_Asking Code=[1] of {mytype};
chan AutomateBilletClient_Asking Code=[1] of {mytype};
proctype Client(){
AutomateBilletClient_Asking_postcard?Asking_postcard;
ClientAutomateBillet_card!card;
AutomateBilletClient_Asking_code?Asking_postcode;
if
 ::(rightcode)-
>ClientAutomateBillet_verifySUM!verifySUM;
 ::Else-
>AutomateBilletClient_Asking_code?Asking_postcode;
fi;}
proctype AutomateBillet(){
AutomateBilletClient_Asking_postcard!Asking_postcard;
ClientAutomateBillet_card?card;
AutomateBilletClient_Asking_code!Asking_postcode;
if
 ::(rightcode)->
ClientAutomateBillet_verifySUM?verifySUM;
 ::Else-
>AutomateBilletClient_Asking_code!Asking_postcode;
fi;}
init{
 ::(true) -> guard=true;
 ::(false) -> guard=false;
}
```

Listing 11

7. Conclusion

We present in this paper a technique to translate UML 2.0 sequence diagrams to PROMELA code. We have shown how concrete syntax-based graph transformation rules can be used to specify a transformation implemented in the software tool AToM^3. This code-generating tool, developed in Python, relies on graph grammars and meta-modeling techniques. It is a great advantage that the developer can specify rules in the well known concrete syntax of sequence diagrams instead of the complicated abstract syntax. Since it takes into account the most popular UML combined fragments, this approach allows the developer to detect flaws in more completed complex sequence diagrams.

We have presented AToM^3, a tool which implements the concepts presented before, and demonstrated its usefulness by generating a PROMELA code for use in SPIN model checker to simulate the execution and to verify properties. In a future work, we plan to transform other combined fragment to PROMELA code. We plan also to perform some verification of properties using SPIN.

References

[1] OMG, Unified Modeling Language Specification Version 1.5, 2003.

[2] J. de Lara and H. Vangheluwe, AToM^3: A Tool for Multi-Formalism and Meta-Modeling, Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering,Grenoble, France (2002),pp. 174–188.

[3] 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.

[4] 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).

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

[6] 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.

[7] 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.

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

[9] Frank Hermann. Typed Attributed Graph Grammar for Syntax Directed Editing of UML Sequence Diagrams. Diploma thesis, Master's thesis, Technical University of Berlin, Department for Computer Science, 2005.

[10] S. Leue 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.

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

[12]Ø. Haugen 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.

[13]V. Lima, C. Talhi "Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of messages", Elsevier 2009.

[14] R. Gronmo and B. Moller-Pedersen "From UML to Sequence Diagram to State Machines by Graph Transformation", 2011.

[15] 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.