Computing and Informatics, Vol. 41, 2022, 1284–1309, doi: 10.31577/cai_2022_5_1284

A GRAPH TRANSFORMATION APPROACH FOR MODELING AND VERIFICATION OF UML 2.0 SEQUENCE DIAGRAMS

Houda Hamrouche

Department of Computer Science
20 Aout 1955 University, BP-26, Skikda, 21000
Algeria
e-mail: h.hamrouche@univ-skikda.dz

Allaoua Chaoui

MISC Laboratory, Department of Computer Science and Its Applications
Faculty of NTIC
University of Constantine 2-Abdelhamid Mehri
BP-67A, Constantine, 25000, Algeria
e-mail: allaoua.chaoui@univ-constantine2.dz

Smaine Mazouzi

LICUS Laboratory, Department of Computer Science
20 Aout 1955 University, BP-26, Skikda, 21000
Algeria
e-mail: s.mazouzi@univ-skikda.dz

Abstract. Unified Modeling Language (UML) 2.0 Sequence Diagrams (UML 2.0 SD) are used to describe interactions in software systems. These diagrams must be verified in the early stages of software development process to guarantee the production of a reliable system. However, UML 2.0 SD lack formal semantics as all UML specifications, which makes their verification difficult, especially if we are modeling a critical system where the automation of verification is necessary. Communicating Sequential Processes (CSP) is a formal specification language that is

suited for analysis and has many automatic verification tools. Thus, UML and CSP have complementary aspects, which are modeling and analysis. Recently, a formalization of UML 2.0 SD using CSP has been proposed in the literature; however, no automation of that formalization exists. In this paper, we propose an approach on the basis of the above formalization and a visual modeling tool to model and automatically transform UML 2.0 SD to CSP ones; thus, the existing CSP model checker can verify them. This approach aims to use UML 2.0 SD for modeling and CSP and its tools for verification. This approach is based on graph transformation, which uses AToM^3 tool and proposes a metamodel of UML 2.0 SD and a graph grammar to perform the mapping of the latter into CSP. Failures-Divergence Refinement (FDR) is the model checking tool used to verify the behavioral properties of the source model transformation such as deadlock, livelock and determinism. The proposed approach and tool are illustrated through a case study.

Keywords: Unified Modeling Language 2.0, Hoare's communicating sequential processes, graph grammar, meta-modeling, model checker, AToM^3 tool

1 INTRODUCTION

The Object Management Group [1] specification of sequence diagrams consists of a concrete and an abstract syntax given with a metamodel. The first one defines the graphical notation, whereas the second aspect describes the relation between the diagram's elements. Unified Modeling Language (UML) 2.0 Sequence Diagrams [2] (UML 2.0 SD) are used to describe interactions in software systems. These diagrams used ordered messages in time, which are structured sometimes in combined fragments, to show the interactions between objects and their occurrence order. This information is conveyed through the horizontal and the vertical axes. An UML 2.0 SD represents the details of an UML use case and models the logical flow of a simple or complex operation. Accordingly, these diagrams allow us to describe the expected running of the software implementation.

Model verification is the process of validating the behavioral properties of an implementation model. Thus, UML 2.0 SD must be verified in the early stages of software development to guarantee the production of a reliable system. However, UML 2.0 SD lack formal semantics, similar to all UML specifications, which makes their verification difficult, especially if we are modeling a critical system where the automation of verification is necessary.

Communicating Sequential Processes [3] (CSP) is a formal specification language that is suited for describing interaction patterns in concurrent systems and has many automatic verification tools, such as: FDR4 [4], PAT [5], and ProB [6]. This language belongs to the family of process algebra and offers tools to describe communication and synchronization between processes. Thus, UML and CSP have complementary aspects, which are modeling and analysis.

The most important reference where UML 2.0 SD are translated into CSP is that of [7], where the objective was to provide concise definitions of the patterns of behaviour modeled by the different interaction operators and using the process algebra CSP. However, no automation of this formalization has been developed. To surmount that limitation, we propose in this study an automatic approach inspired from [7] and a visual modeling tool to model and automatically transform UML 2.0 SD into CSP_M ready for execution with FDR4 model checker. So, this approach, based on graph transformation, aims to use UML 2.0 SD for modeling and CSP and its tools for analysis.

There are three basic steps to achieve our goal:

Metamodeling of UML 2.0 SD: In this step, we define a metamodel of UML 2.0 SD using AToM^3 tool, then generate the visual modeling tool. Constraints are written in Python.

Automatic transformation of UML 2.0 SD into CSP expressions: In this second step, we define the graph grammar to automatically transform UML 2.0 SD into CSP_M. Actions and constraints of the transformation rules are implemented in Python. The left hand side and the right hand side of each rule are created in the graphical environment previously generated.

Verification: In this third step, we define CSP_M assertions and use FDR4 model checker to verify the behavioral properties of the source model transformation.

The proposed approach covers all message types (synchronous, asynchronous, and acknowledgement) and formalizes the following interaction operators of combined fragments: Seq, Strict, Par, Alt, Opt, Break, Loop, Ignore, Consider, and Assert. The remainder of the paper is organized as follows. In Section 2, we survey related works. In Section 3, we review some basic concepts of the field of study. In Section 4, we describe the proposed approach that transforms UML 2.0 SD to CSP code. More precisely, we show the proposed metamodel and the graph grammar that allows the mapping between the source and the target models of the transformation. In Section 5, we illustrate the approach through a case study. In Section 6, we conclude the paper and provide an outlook regarding future work.

2 RELATED WORK

In the literature, several studies attempt to formalize UML 2.0 SD in CSP. In [7], the authors provided UML 2.0 SD a CSP semantic in an implementation independent fashion; they viewed sequence diagrams in terms of occurrence observations in place of messages. In addition, their work covers all interaction operators. [8] proposed an automatic translation of UML 2.0 SD into CSP by using XSLT [9] programming language. The proposed approach has many restrictions, such as the formalization of combined fragments with only four interaction operators: Opt, Break, Loop, and Alt. [10] defined new CSP operators for the synthesis of sequence diagrams to verify the correctness property; however, it passes over the combined fragments.

[11] generated a CSP code from sequence diagrams and developed a verification tool by using the PAT model checking tool to verify them. If a counter example is found, then it can be translated back to a sequence diagram for correction. Nevertheless, this approach does not treat combined fragments.

UML 2.0 SD are automatically verified after translation into labeled generalized stochastic Petri net [12] models in [13], colored Petri nets [14] in [15] and [16], and Petri nets in [17].

[18] showed a translation of sequence diagrams to Promela [19] codes to verify the consistency between sequence and statechart diagrams. However, [20] and [21] proposed a formal verification and validation technique for sequence diagrams. They created a Promela code and used the Spin [22] model checker to verify properties written in linear temporal logic (LTL).

[23] developed a tool supporting the transformation of sequence diagrams to Event B [24]. [25] showed the use of multi layer transformations to generate Büchi automata [26] from UML 2.0 SD. [27] defined a method for translating UML sequence diagrams annotated with MARTE stereotypes to time Petri nets with action duration specifications.

Several studies focus on formalizing other UML diagrams in CSP. State diagrams are given a CSP semantic in [28], where activity diagrams are formalized in CSP in [29] and [30]. This last formalization has been implemented in [31] by using diverse graph transformation tools, in [32] by using AToM^3 tool, and in [33] by using AToMPM [34] tool for meta-modeling and GROOVE [35] tool for transformation and properties verification. [36] proposed a graph transformation approach to generate Maude specifications from UML activity diagrams using AToM^3 tool.

The subject of verifying model transformations was discussed in [37, 38, 39, 40]. [41] verified the model transformations by using Isabelle/HOL [42] and Scala [43].

3 BACKGROUND

3.1 UML 2.0 SD

The sequence diagram is one of the important UML interaction diagrams, and it allows representing exchanges between the different objects and actors of the system in the function of time. If the system to be modeled is complicated, then we cannot model the overall dynamics of the system in a single diagram. Therefore, we will call upon a set of sequence diagrams each corresponding to a subfunction of the system. Combined fragment is a part of the sequence diagram that defines a combination of interaction fragments. This mechanism allows the user to describe a number of traces in a compact and concise manner. A combined fragment contains interaction operands and operator, which define the message execution order. Figure 1 represents an example of UML 2.0 SD.

Table 1 shows interaction operator kind values (IOK), designations (Dsgn) and descriptions.

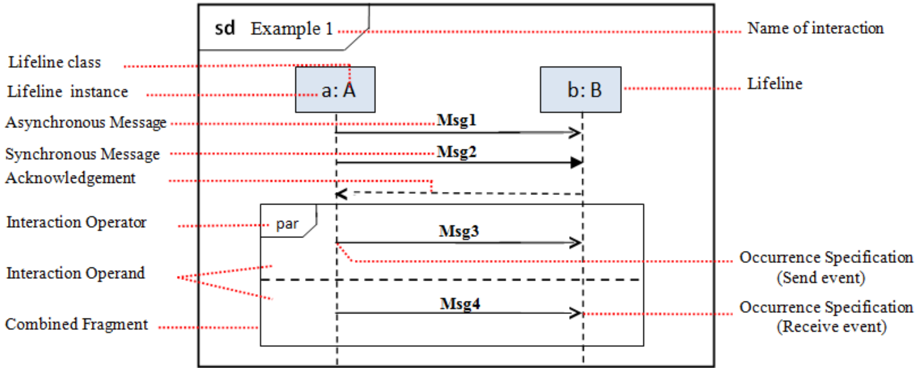

Figure 1. An example of UML 2.0 SD

3.2 CSP

CSP is a notation for describing and analyzing concurrent systems. This language has many extensions with the corresponding supporting tools, such as FDR, CSP++ [44] and ProB for CSP_M [45] and PAT for CSP# [46]. The user can choose the extension and the appropriate verification tool according to the system characteristics [47]. In CSP, processes are independent entities that can communicate with each other. A process can execute events that describe the behavior of processes. The set of events that process P can perform is called its alphabet. The interactions between processes or with its environment are described using a set of process algebraic operators. CSP operators sufficient to describe sequence diagrams can be summarized in the following expression (the notations used are: P, Q: process; X, Y: sets of events; e: event; and b: Boolean condition):

$$
\begin{aligned}
P \mathrel{\hat{=}} (P|Stop|Skip|e \to P|\text{if } b \text{ then } P \text{ else } Q|P[]Q| \\
P| \sim |Q|P \backslash X|P;Q|P[X||Y]Q|P[|X|]Q|P|||Q) \,. \qquad (1)
\end{aligned}
$$

The description of these operators is shown in Table 2.

FDR provides a number of replicated operators that allow the composition of the processes using the same operator. In Table 3, we present a brief description of some of these operators.

3.3 Graph Transformation

Model-Driven Architecture (MDA) [48] is a framework based on many industry standards for visualizing, storing, and exchanging software designs and models. MDA offers another way of software development, and it is an approach to use models instead of the traditional source code. Portability, interoperability, and re-usability

IOK	Dsgn	Description
Alt	Alternative	Combined fragment represents a choice of behavior. The chosen operand must have its guard expression evaluated to true (one operand at most will be chosen).
Opt	Option	Combined fragment represents choice of behavior, and has only one operand. If the guard expression is evaluated to true the operand happens else nothing happens.
Break	Break	Combined fragment has one operand and represents breaking scenario. If the guard expression is true, the operand is chosen and the rest of the enclosing interaction is ignored. Else, the operand is ignored and the rest of the enclosing interaction is chosen.
Par	Parallel	Behaviors of the operands within this combined fragment can be executed in parallel.
Seq	Weak Sequencing	1. In each operand, the ordering of occurrence specifications is maintained in the result. 2. Occurrence specifications within different operands on different lifelines may come in any order. 3. On the same lifeline, occurrence specifications of the first operand comes before that of the second operand.
Strict	Strict Sequencing	This interaction operator defines strict ordering of the operands.
Neg	Negative	Combined fragment represents a set of invalid traces.
Critical	Critical	Combined fragment represents a critical region that it traces cannot be interleaved by other occurrence specifications.
Ignore/ Consider	Ignore/ Consider	"Ignore" interaction operator designates that some message types can be considered as insignificant, so they are ignored. At the other hand, "Consider" interaction operator designates considered messages in the combined fragment.
Assert	Assertion	In this case the only valid continuations are the sequences of the assertion operand.
Loop	Loop	The operand will be iterated a number of times and the loop will terminate if the guard is evaluated to false.

Table 1. Interaction operator kind of UML 2.0 SD combined fragment

are the main goals of MDA. Models can be accessed and automatically transformed by tools for various platforms [49]. The MDA model transformation is conducted by mapping an initial model to an equivalent target model. Each model must be described by a metamodel, which identifies the characteristics of this model. Then, the mapping is defined as a translation between the initial and the target metamodels. Graph grammars [50] are a generalization of Chomsky grammars for graphs. These grammars are composed of rules or productions of the left and right hand sides: $P = (\mathrm{LHS}, \mathrm{RHS})$. The application of each rule allows the transformation of an initial graph called host graph by replacing one of its parts by another graph. The main problem of this replacement is how to relate RHS to the context of the

Expression	Description
Stop	Deadlock CSP process. Denotes the most simple behavior, it does nothing.
Skip	Process that models successful termination.
$e \rightarrow P$	Prefix: Performs the event e and then behaves as the process P.
if b then P else Q	Guarded expression: Behaves as P if the Boolean guard b is true, and as Q otherwise.
$P[]Q$	External (deterministic) choice: offers the environment to choose between the initial events of P and Q.
$P \vert \sim \vert Q$	Internal (nondeterministic) choice: choose one of P and Q, and then run the chosen process.
$P \setminus X$	Hide: behaves like P, but the events of X are hiden and turned into internal events.
$P; Q$	Sequential composition: behaves as P until its successful termination, then behaves as Q.
$P[X \vert\vert Y]Q$	Alphabetised Parallel: synchronised parallel run of P and Q on the set of events $X \cap Y$.
$P[\vert X \vert]Q$	Generalised Parallel: synchronised parallel run of P and Q on events in X.
$P \vert\vert\vert Q$	Interleave: unsynchronised parallel run of P and Q.

Table 2. Description of CSP operators

target graph. Figure 2 shows the rule-based modification of graphs associated to the model transformation architecture.

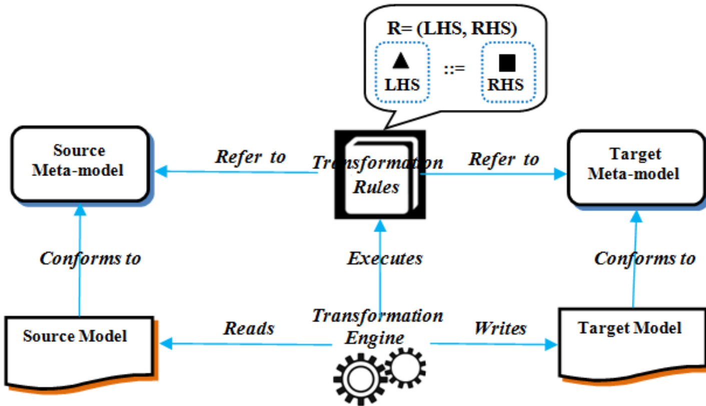

Figure 2. Model transformation architecture

Expression	Description	If composition result is empty
$[]\langle \text{set statements}\rangle @P$	Replicated External Choice: for each value of the statements, P is evaluated then the resulting processes are composed using [] operator.	Return STOP.
$\vert \sim \vert\langle \text{set statement}\rangle > @P(x)$	Replicated Internal Choice: for each value of the statements, P is evaluted then the resulting processes are composed using $\vert \sim \vert$ operator.	Display error.
$\vert\vert\langle \text{set statements} > @[A]P$	Replicated Alphabetised Parallel: for each value of the statements, P and its alphabet A are evaluated then the resulting processes are composed using alphabetised Parallel operator.	Return SKIP.
$[\vert X\vert]\langle \text{set statements} > @P$	Replicated Generalised Parallel: for each value of the statements, P is evaluated then the resulting processes are composed and Synchronise on X using Generalised Parallel operator.	Return SKIP.
$\vert\vert\vert\langle \text{set statements}\rangle @P$	Replicated Interleave: for each value of the statements, P is evaluated then the resulting processes are composed using Interleave operator.	Return SKIP.

Table 3. Description of CSP replicated operators

3.4 AToM^3

AToM^3 is a visual tool for modeling and metamodeling multiformalisms. This tool relies on rewriting graphs that use graph grammar rules for the definition of the transformations between the formalisms and the generation of code and the specification of simulators. The user must define the rules (LHS, RHS), priorities, and conditions. The two main tasks of AToM^3 are metamodeling and model transformation. The main metaformalisms used are Entity-Relation and UML class diagram formalism. [51] showed a survey and classification of existing metamodel-based transformation tools and a comparison between them using a qualitative framework. Among the Graph-based tools, we can cite TGG-INTERPRETER [52], AGG [53], and GreAT [54].

4 THE PROPOSED APPROACH

In this study, we adopt from [7] the idea of mirroring the structure of a sequence diagram. In this formalization of UML 2.0 SD in CSP, each lifeline is mapped to a process named Lifeline(). Each message is mapped to a process Message(), and every occurrence observation is mapped to a CSP event. Sending and receiving messages are detached, and a message cannot be received before it was sent. The temporal order of the occurrence observations on the corresponding lifeline is respected using process PrefixComposition(). Strict sequencing is defined using an additional process Enforce(), which guarantees the message order over all participating lifelines. Lifelines() and Messages() processes model the parallel composition of PrefixComposition() and Message() processes, respectively. Figures 3 and 4 show the formalization of Message() and PrefixComposition() processes, respectively.

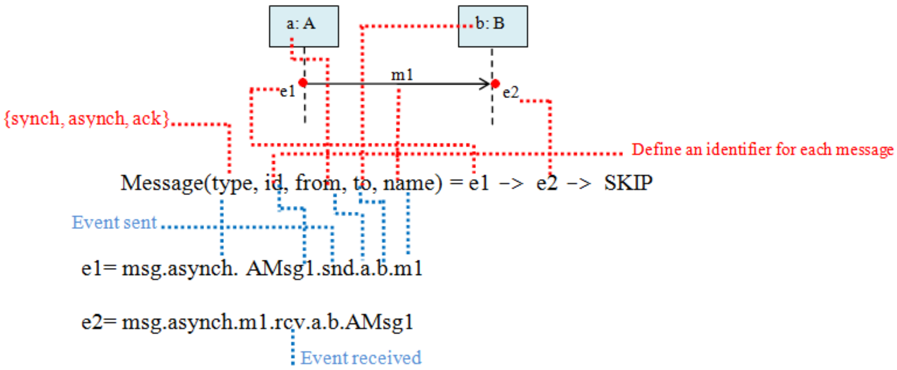

Figure 3. Formalization of Message() process

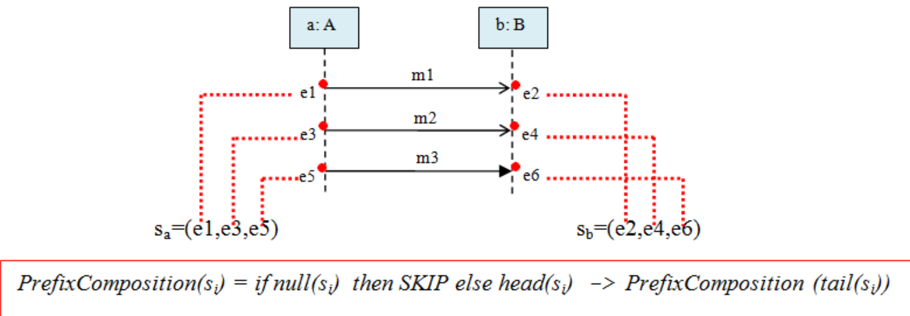

Figure 4. Formalization of PrefixComposition() process

To have a better understanding of UML 2.0 SD adopted formalization, the reader can be referred to [7].

We propose a metamodel of UML 2.0 SD and a graph grammar to reach the goal of transforming UML 2.0 SD into CSP code. CD-ClassDiagramsV3 metaformalism and AToM^3 tool are used to perform the modeling and the transformation of UML 2.0 SD. FDR is a model checking tool used to verify the behavioral properties of the source model transformation. Figure 5 shows the architecture of the proposed approach.

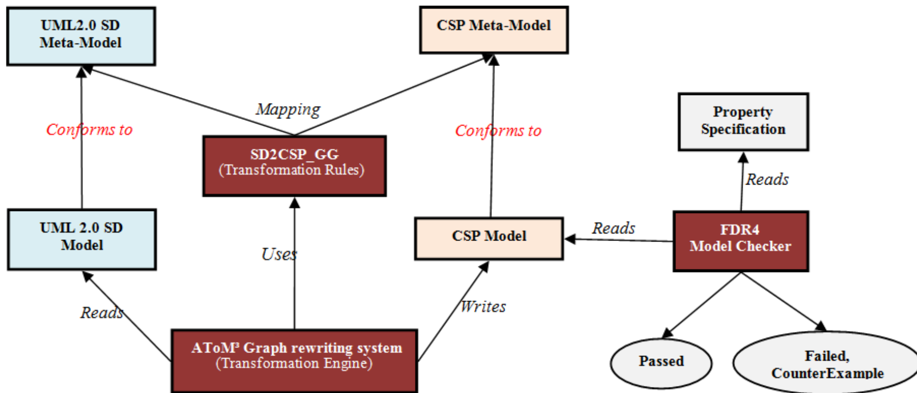

Figure 5. Architecture of the proposed approach

The steps of our approach are as follows:

1. Metamodeling of UML 2.0 SD
   (a) Define the metamodel of UML 2.0 SD (SDiagram_META).
   (b) Generate the visual modeling tool for modeling UML 2.0 SD models.
2. Graph grammar. Define the graph grammar (SD2CSP_GG) to transform UML 2.0 SD into CSP_M.
3. Verification
   (a) Define CSP_M assertions.
   (b) Use an FDR4 model checker to verify the CSP_M program by using CSP_M assertions. Then, generate the verification result (passed if the property is satisfied and failed plus counter example otherwise).

4.1 Metamodeling of UML 2.0 SD

Given that the AToM^3 tool is used to describe formalisms and generate tools, we used the metaformalism CD-ClassDiagramsV3 to create the metamodel (SDiagram_META) to process UML 2.0 SD. This last diagram consists of interactions,

interaction operands, combined fragments, and lifelines with ordered messages between them. The proposed metamodel contains four classes linked by six associations. Constraints and actions are expressed in Python [55] code. The visual representation of each class or association is specified in this step. The metamodel is composed of the following classes:

Interaction: This class has a Name and represents an interaction in the sequence diagram.

CombinedFragment: This class describes a combined fragment and has two attributes: Name and IOKind, which represents the interaction operator kind.

InteractionOperand: This class describes an interaction operand and has two attributes: Name and IOConstraint, which represents the interaction operand constraint.

LifeLine: This class depicts a lifeline and has two attributes, namely, Name and ClassName. These classes are linked by the following associations:

IContains: This association exists between an interaction and a lifeline or combined fragment. The concept expresses the notion of hierarchy: lifelines and combined fragments are inside an interaction.

CFContains: This association expresses the notion of hierarchy between a combined fragment and its interaction operands.

IOContains: This association expresses the notion of hierarchy between an interaction operand and lifelines and combined fragments (in the case of nested combined fragments, the interaction operand has combined fragments inside it).

AMessage: Represents an asynchronous message between two lifelines.

SMessage: Represents a synchronous message between two lifelines.

Acknowledgement: Represents an acknowledgement between two lifelines.

The proposed metamodel is shown in Figure 6.

The graphical environment for modeling sequence diagram models generated from SDiagram_META by AToM^3 tool with a dialog box to edit an asynchronous message is presented in Figure 7.

The proposed approach covers the interaction operators of combined fragment shown in Figure 8.

4.2 Graph Grammar for the Transformation of UML 2.0 SD to CSP Expressions

The researchers have proposed a graph grammar that contains eleven rules, initial action, and final action to generate a CSP code from sequence diagrams. The generated code is written in CSP_M. The machine readable dialect of CSP is the accepted input type by the FDR4 tool. Figure 9 shows the graphical representation of the rules where their description is as follows:

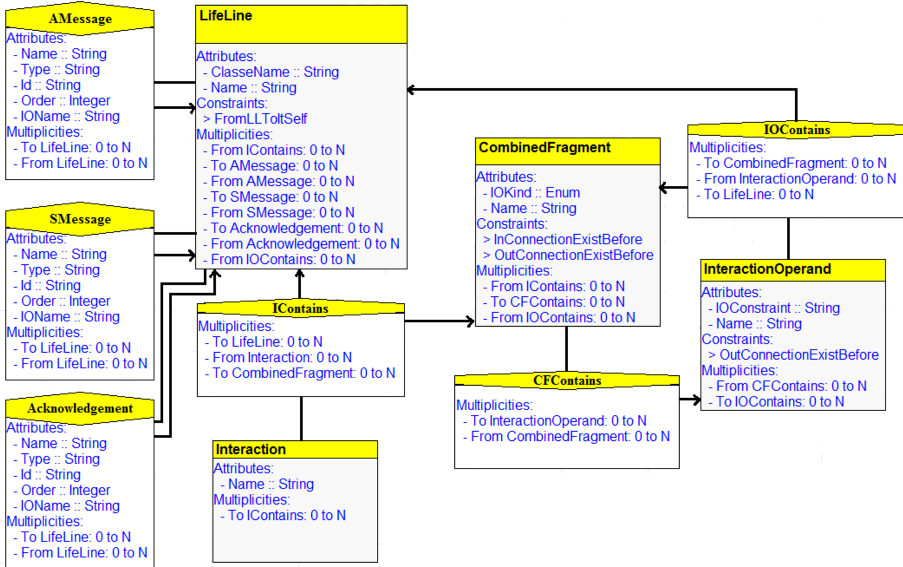

Figure 6. Sequence diagram metamodel

Initial Action: The role of the initial action is to open the file where the CSP code will be generated and to create temporary attributes to be used in the rules.

Rule 1: rule_AMessage (Priority 1): This rule is applied to locate an asynchronous message that has not been previously processed, and it generates: event sent, event received, and Message() process.

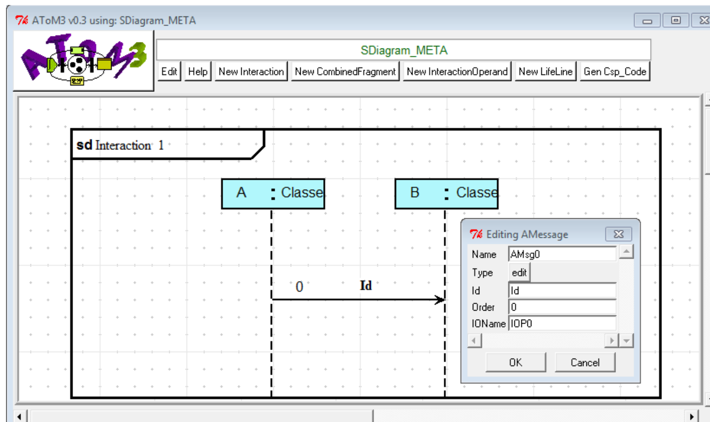

Figure 7. Sequence diagram tool

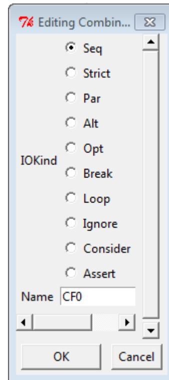

Figure 8. Interaction operators

Rule 2: rule_SMessage (Priority 1): It is applied to locate a synchronous message that has not been previously processed, and it generates two events and a process: event sent, event received, and Message() process.

Rule 3: rule_Acknowledgement (Priority 1): It is applied to locate an acknowledgement that has not been previously processed, and it generates: event sent, event received, and Message() process.

Rule 4: rule_IOcontainsLLam (Priority 2): It generates PrefixComposition() process for both lifelines: sender and receiver of an asynchronous message.

Rule 5: rule_IOcontainsLLsm (Priority 2): It generates PrefixComposition() process for the lifelines: sender and receiver of a synchronous message.

Rule 6: rule_IOcontainsLLack (Priority 2): It generates PrefixComposition() process for the lifelines: sender and receiver of an acknowledgement.

Rule 7: rule_IOcontainsCF (Priority 3): It is applied to locate the combined fragment, which is a child of an interaction operand.

Rule 8: rule_CFcontainsIO (Priority 4): It is applied to locate an interaction operand (not previously processed) and generates processes: Messages(), Lifelines(), Seq(), and Strict().

Rule 9: rule_NestedCF (Priority 5): It is applied to locate a complex interaction that contains two nested combined fragments.

Rule 10: rule_IcontainsCF (Priority 6): This rule is applied to generate the adequate process to the interaction operator kind (IOKind).

Rule 11: rule_Interaction (Priority 7): It generates the principal process P of the principal interaction.

Final Action: The role of the final action is to erase the temporary attributes and close the output file.

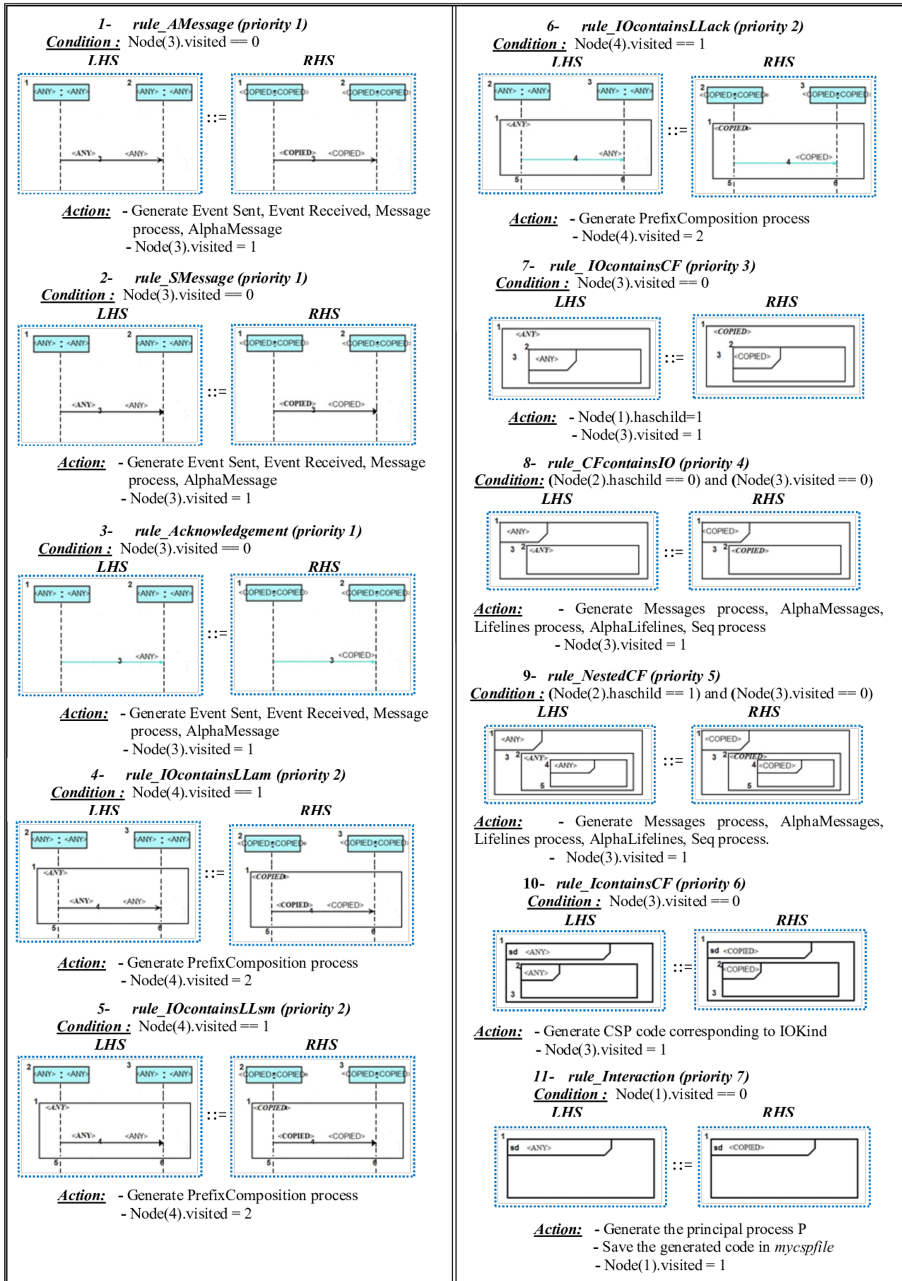

Figure 9. Graph grammar to transform UML 2.0 SD into CSP expressions

We only present in detail the first rule of the graph grammar, as shown in Figure 10, due to the space constraint.

Condition : locate an *asynchronous* message that has not been previously processed.

```
node=self.getMatched(graphID, self.LHS.nodeWithLabel(3))
return node.visited == 0
```

Action: Generate event sent, event received and *Message()* process for an *asynchronous* message.

```
import string
node1=self.getMatched(graphID,self.LHS.nodeWithLabel(1))
node2=self.getMatched(graphID, self.LHS.nodeWithLabel(2))
node3=self.getMatched(graphID, self.LHS.nodeWithLabel(3))
sender = node1.Name.getValue()
receiver = node2.Name.getValue()
message = node3.Name.getValue()
id = node3.Id.getValue()
O = node3.Order.getValue()
Iop = node3.IOName.getValue()
es=msg.asynch.'+id+'.snd.'+sender+'.'+receiver+'.'+message
ess='msg.asynch.'+id+'.snd.'+sender+'.'+receiver+'.'+ message +':' + iop
er= 'msg.asynch.'+id+'.rcv.'+sender+'.'+receiver+'.'+message
err='msg.asynch.'+id+'.rcv.'+sender+'.'+receiver+'.'+message+':'+iop
node1.listS= es+':'+er
node3.m=(asynchronous,'+id+','+sender+','+receiver+','+ message+')'
node3.PMsg= 'Message(asynchronous,'+id+','+sender+','+receiver+','+mes sage+')='+es+' -> '+
er+' -> skip'
node3.AMsg='AlphaMessage(asynchronous,'+id+','+sender+','+receiver+','+message+')={|'+es+','+
er+'|}'
pmsg = node3.PMsg
amsg = node3.AMsg
node1.listS.insert(0,ess)
node2.listS.insert(0,err)
node3.visited  = 1
```

Figure 10. Python code of an asynchronous message transformation

4.3 Verification

We can distinguish three ways to obtain the mathematical understanding of the meaning of a CSP program. These ways are algebraic, operational, and denotational semantics.

The different denotational semantics for CSP are based on traces, failures, and divergences [56]. The traces model associates with each process P the finite sequences of events allowed by this process. Thus, this model enables us to represent the possible behaviors of processes in the form of traces. Traces(P) denote the traces of process P.

The stable failures model of process P, denoted by failures(P), associates with each process P couples of the form (s, X), where s is a finite trace admitted by P, and X is the set of events that cannot be executed after executing the events of s.

Finally, the failures-divergences model associates with each process P all of its stable failures and divergences. Process P is divergent if it is in a state in which the only possible events are the internal events. This state is said to be divergent. The set of divergences(P) is the set of traces s; accordingly, the process is found in a divergent state after performing s. The refinement concept consists of calculating and comparing the semantic models of two processes.

FDR4 is a refinement checker for CSP designed to analyze models written in CSP_M. The machine readable dialect of CSP combines Hoare's CSP operators with a functional programming language. The primary aim is not to describe algorithms

in an executable form but to support the description of parallel systems in an automatic manipulation form. FDR4 allows various forms of assertions to check the properties of deadlock, livelock, determinism, and specified partial order reduction and refinement assertions.

If P and Q are two processes, and S is one of the three semantic models (T: Traces, F: Failures, FD: Failures-Divergences), then FDR can simply check if Q refines P ($P \sqsubseteq_S Q$) by inserting the following assertion in the CSP_M code: assert $P[s = Q$.

We can use trace refinement model to check the safety and stable failures model to verify the liveness property.

4.4 Illustrative Example

a. Presentation

This interaction shows the communication between two lifelines "S" and "R" through an asynchronous message "m1". It is part of an interaction operand which belongs to the combined fragment "Seq". Figure 11 presents this interaction created in our tool.

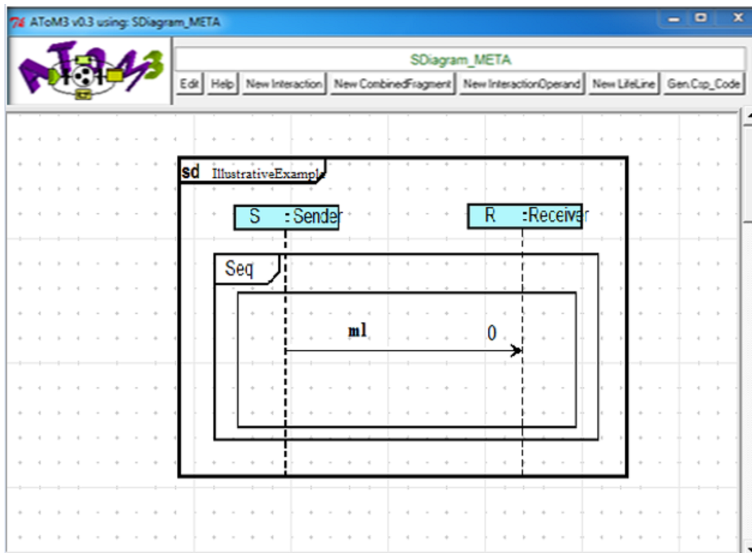

Figure 11. Illustrative example created in our tool

b. Translating sequence diagram to CSP expressions

The execution of SD2CSP_GG graph grammar on the interaction presented in Figure 11 leads to the execution of the following rules: Initial Action,

rule_Amessage, rule_IOcontainsLLam, rule_CFcontainsIO, rule_IcontainsCF, rule_Interaction and Final Action. The obtained result is presented in Figure 12.

```
1.  datatype NAME = m1
2.  datatype MTYPE= snd|rcv|ack
3.  datatype LLINE= S|R
4.  datatype M=AMsg0
5.  datatype te= synch.M.MTYPE.LLINE.LLINE.NAME| asynch.M.MTYPE.LLINE.LLINE.NAME
6.  channel msg:te
7.  p=Seq((<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>),(<(asynchronous,AMsg0,S,R,m1)
8.  >))
9.  Seq((<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>),(< (asynchronous,AMsg0,S,R,m1)>
10. ))=Lifelines(<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch. AMsg0.rcv.S.R.m1>)
11. [alphaLifelines(<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>)||alphaMessages(<
12. (asynchronous,AMsg0,S,R,m1)>)]Messages(<(asynchronous,AMsg0,S,R,m1)>)
13. Lifelines(<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>)= ||line:
14. {<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>}@[set(line)] PrefixComposition(line)
15. alphaLifelines(<msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1>)=
16. {|msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.m1|}
17. PrefixComposition(s)= if null(s) then SKIP else head(s) -> PrefixComposition(tail(s))
18. Messages(<(asynchronous,AMsg0,S,R,m1)>)=||(t,id,from,to,n):{(asynchronous,AMsg0,S,R,m1)}@
19. [alphaMessages(asynchronous,AMsg0,S,R,m1)]Message(t,id,from,to,n)
20. alphaMessages(<(asynchronous,AMsg0,S,R,m1)>)= alphaMessage(asynchronous,AMsg0,S,R,m1)
21. Message(asynchronous,AMsg0,S,R,m1)= msg.asynch.AMsg0.snd.S.R.m1 -> msg.asynch.AMsg0.rcv.S.R.m1
22. -> SKIP
23. alphaMessage(asynchronous,AMsg0,S,R,m1)={|msg.asynch.AMsg0.snd.S.R.m1,msg.asynch.AMsg0.rcv.S.R.
24. m1|}
25. assert p:[deadlock free[F]]
26. assert p:[divergence free [FD]]
27. assert p:[deterministic[FD]]
```

Figure 12. Generated CSP code

In the following, we give in details the results of the execution of each rule.

Initial Action: Opens the file mycspfile.csp to save the generated CSP code and creates temporary attributes.

rule_Amessage: The execution of this rule generates:
Event sent: msg.asynch.AMsg0.snd.S.R.m1,
Event received: msg.asynch.AMsg0.rcv.S.R.m1,
Message() parameters: asynchronous, AMsg0, S, R, m1,
Message() process: lines (21, 22) and Message() alphabet: lines (23, 24).

rule_IOcontainsLLam: This rule generates PrefixComposition() process: line (17).

rule_CFcontainsIO: This rule generates Messages() process in lines (18, 19), Messages() alphabet in line (20), Lifelines() process in lines (13, 14) and Lifelines() alphabet in lines (15, 16).

rule_IcontainsCF: This rule generates the adequate process to Seq operator. It generates Seq() process from the parallel composition of Messages() and Lifelines() processes, the corresponding code is presented in lines (9...12).

rule_Interaction: This rule generates the principal process P, as shown in lines (7, 8), declares datatypes and msg chanel in lines (1...6).

Final Action: This rule erases the temporary attributes and closes the output file. CSP_M assertions stated in lines (25...27) will be explained in Section 5, part c.

c. Verification

We applied FDR4 to the CSP$_M$ specification stated in Figure 12 and we got the result shown in Figure 13.

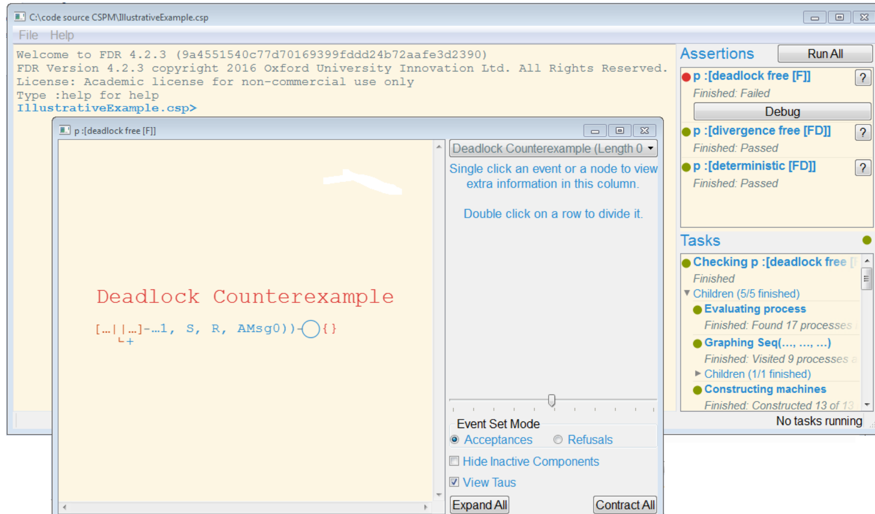

Figure 13. Illustrative example verification using FDR4

5 CASE STUDY

a. Presentation

In this case study we create, transform and verify UML 2.0 SD of an Automated Teller Machine (ATM). It is a modified version of a tradition case study proposed in [21]. This interaction consisting of three objects: The user, the ATM and the bank, and twenty two messages passed between these objects and organized in seven combined fragments. It shows inserting card, verifying card and personal identification number (PIN), ejecting card and performing some operations. Figure 14 shows this interaction created in our tool.

b. Translating ATM sequence diagram to CSP code

The execution of SD2CSP_GG graph grammar on the ATM sequence diagram presented in Figure 14, provides a CSP code. A part of the obtained result is presented in Figure 15.

c. Verification

We applied FDR4 to the CSP$_M$ specification stated in Figure 15. and obtained the result shown in Figure 16. Deadlock, livelock and determinism properties were checked as follows:

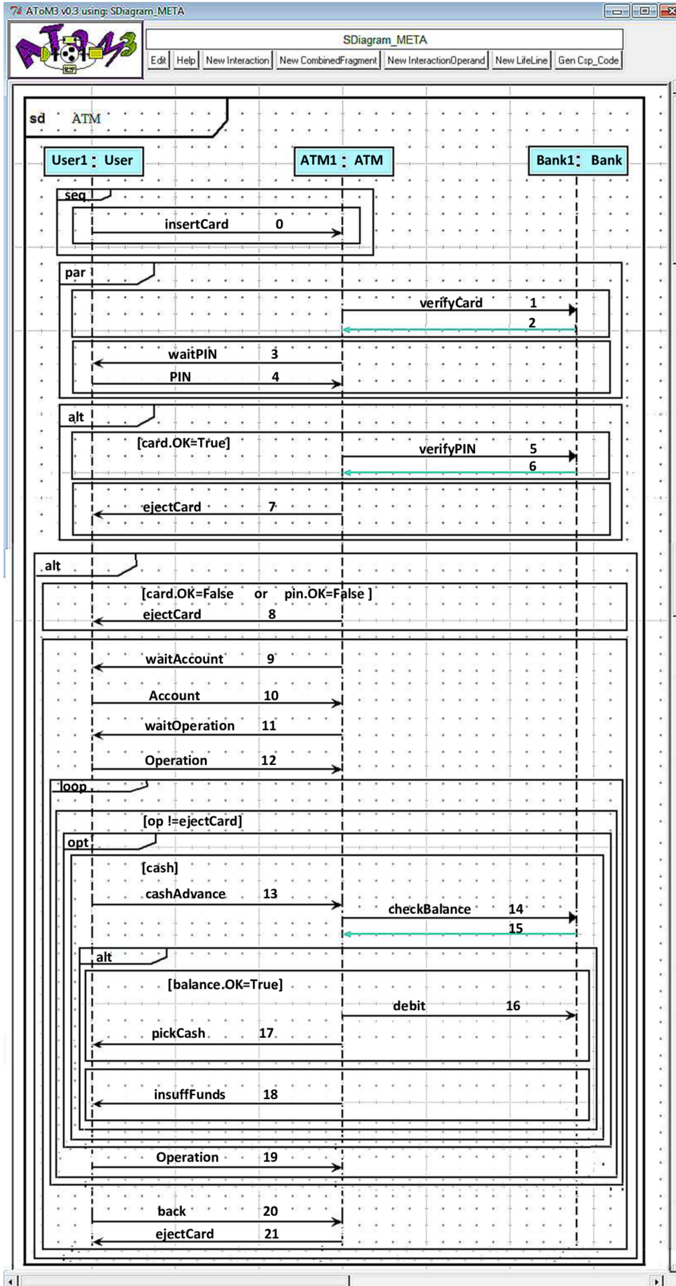

Figure 14. ATM sequence diagram created in our tool

```
datatypete1=OK|NOTOK
datatypeNAME=insertCard|verifyCard|waitPIN|PIN|verifyPIN
|ejectCard|waitAccount|Account|waitOperation|Operation|
cashAdvance|checkBalance|debit|pickCash|insuffFunds|back
datatypeMTYPE=snd|rcv|ack
datatypeLLINE=User1|ATM1|Bank1
datatypeM=AMsg0|AMsg1|AMsg2|AMsg3|AMsg4|AMsg5|AMsg6|AMsg7|AMsg8|AMsg9|AMsg10|AMsg11|AMsg12
|AMsg13|AMsg14|AMsg15|AMsg16|AMsg17|AMsg18|AMsg19|AMsg20|AMsg21
datatypete=synch.M.MTYPE.LLINE.LLINE.NAME| asynch.M.MTYPE.LLINE.LLINE.NAME
channel msg:te
channel card,pin,balance:te1
channel op:NAME
channel cash:Bool

p=Seq((<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>>),(<(asynch,AMsg0,User1,ATM1,i
nsertCard)>));par((<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.A
TM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>),(<(synch,AMsg1,ATM1,Bank1,verifyCard)>),(<<msg.asynch.AMsg3.r
cv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1
.ATM1.PIN>>),(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>));Alt((<<msg.synch.AMsg5.snd.ATM1.Bank1.verify
PIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verify
PIN>>),(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>),<card.OK>,(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.AT
M1.User1.ejectCard>>),(<(asynch,AMsg7,ATM1,User1,ejectCard)>));Alt((<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg
8.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg8,ATM1,User1,ejectCard)>),<card.NOTOK,pin.NOTOK>,(<<msg.asynch.AMsg9.rcv.ATM1.Use
r1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.Use
r1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd
.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>),(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,
User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>))
```

Figure 15. Generated CSP code

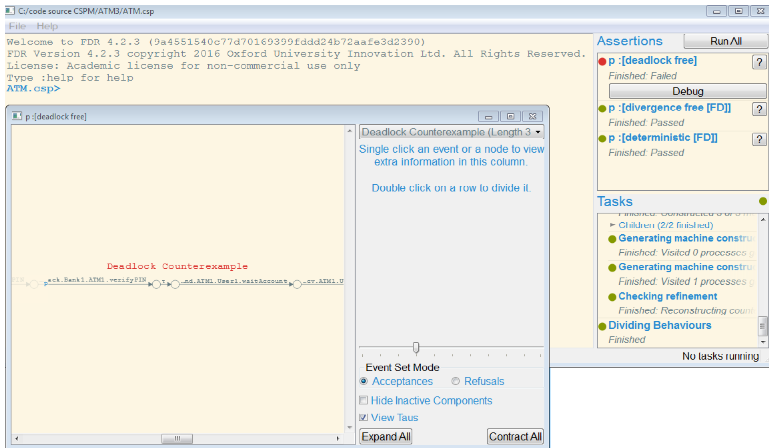

Figure 16. ATM sequence diagram verification using FDR4

Deadlock: In FDR4, to assert that a process P is deadlock free in the failures-divergences model, we can simply write: assert P: [deadlock free], or write: assert P: [deadlock free $[F]$] if we want to check the assertion in the failures model. If the checking is failed, a counter example can be displayed, as shown in Figure 16.

Livelock: We say that a process P can diverge if it executes infinite internal actions (livelock). So, checking that P is a livelock freedom can be stated as: assert P: [divergence free], or assert P: [divergence free $[FD]$].

Determinism: We can check that a process P is deterministic by using one of the following assertions: assert P: [deterministic] or assert P: [deterministic $[FD]$].

The check of deadlock property was failed, but those of livelock and determinism were passed.

6 CONCLUSION

In this study, we have proposed a graph transformation approach and a visual modeling tool to model and automatically transform UML 2.0 SD to CSP expressions using the metamodeling tool AToM^3. To reach this goal, we have proposed a metamodel for UML 2.0 SD and a graph grammar to achieve the transformation. We have used a CD-ClassDiagramsV3 as a metaformalism and Python code to specify constraints and actions. Then, we have performed the verification of some properties such as deadlock, livelock and determinism using the FDR4 model checker. Finally, we have illustrated the approach through a case study.

In future work, we plan to transform other UML diagrams to CSP expressions and verify the equivalence between the source and the target models of the transformation. We will also use other model checkers of CSP, such as PAT and ProB, to deal with the limits of refinement testing of FDR. Finally, we plan to tackle the problem of formal correctness of the transformation itself by using theorem provers, such as Isabelle and Coq [57].

REFERENCES

[1] OMG: Object Management Group. `https://www.omg.org` [accessed 2021-04-17].

[2] UML: Unified Modeling Language. `https://www.omg.org/spec/UML/2.0/` [accessed 2021-04-17].

[3] Hoare, C. A. R.: Communicating Sequential Processes. Communications of the ACM, Vol. 21, 1978, No. 8, pp. 666–677, doi: 10.1145/359576.359585.

[4] FDR: FDR Manual, Release 4.2.7. 2020.

[5] Sun, J.—Liu, Y.—Dong, J. S.—Pang, J.: PAT: Towards Flexible Verification Under Fairness. In: Bouajjani, A., Maler, O. (Eds.): Computer Aided Verification (CAV 2009). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5643, 2009, pp. 709–714, doi: 10.1007/978-3-642-02658-4_59.

[6] Leuschel, M.—Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (Eds.): FME 2003: Formal Methods. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2805, 2003, pp. 855–874, doi: 10.1007/978-3-540-45236-2_46.

[7] Jacobs, J.—Simpson, A.: On a Process Algebraic Representation of Sequence Diagrams. In: Canal, C., Idani, A. (Eds.): Software Engineering and Formal Methods

(SEFM 2014). Springer, Cham, Lecture Notes in Computer Science, Vol. 8938, 2014, pp. 71–85, doi: 10.1007/978-3-319-15201-1_5.

[8] Dan, L.—Danning, L.: An Approach to Formalize UML Sequence Diagrams in CSP. 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010), 2010.

[9] Kay, M.: XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer). 4th Edition. Wrox Press Ltd., 2008.

[10] Kaizu, T.—Isobe, Y.—Suzuki, M.: Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E96.A, 2013, No. 2, pp. 495–504, doi: 10.1587/transfun.E96.A.495.

[11] Kaizu, T.—Isobe, Y.—Suzuki, M.: SDVerifier: A Tool for Verification of Sequence Diagrams Using the Process Algebra CSP. Computer Software, Vol. 32, 2015, No. 1, pp. 234–252, doi: 10.11309/jssst.32.1_234.

[12] Marsan, M. A.—Balbo, G.—Conte, G.—Donatelli, S.—Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Edition 1st. John Wiley and Sons, Inc., 1994.

[13] Bouarioua, M.—Chaoui, A.—Elmansouri, R.: From UML Sequence Diagrams to Labeled Generalized Stochastic Petri Net Models Using Graph Transformation. In: Yonazi, J. J., Sedoyeka, E., Ariwa, E., El-Qawasmeh, E. (Eds.): e-Technologies and Networks for Development (ICeND 2011). Springer, Berlin, Heidelberg, Communications in Computer and Information Science, Vol. 171, 2011, pp. 318–328, doi: 10.1007/978-3-642-22729-5_27.

[14] Jensen, K.—Kristensen, L. M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Edition 1st. Springer, Berlin, Heidelberg, 2009, doi: 10.1007/b95112.

[15] Custódio Soares, J. A.—Lima, B.—Pascoal Faria, J.: Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets. Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development – Volume 1: AMARETTO, SciTePress, 2018, pp. 668–679, doi: 10.5220/0006731806680679.

[16] Mozaffari, M.—Harounabadi, A.: Verification and Validation of UML 2.0 Sequence Diagrams Using Colored Petri Nets. 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 117–121, doi: 10.1109/ICCSN.2011.6013675.

[17] Cunha, E.—Custódio, M.—Rocha, H.—Barreto, R. S.: Formal Verification of UML Sequence Diagrams in the Embedded Systems Context. 2011 Brazilian Symposium on Computing System Engineering (SBESC 2011), IEEE, 2011, pp. 39–45, doi: 10.1109/SBESC.2011.18.

[18] Zhao, X.—Long, Q.—Qiu, Z.: Model Checking Dynamic UML Consistency. In: Liu, Z., He, J. (Eds.): Formal Methods and Software Engineering (ICFEM 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4260, 2006, pp. 440–459, doi: 10.1007/11901433_24.

[19] Promela: Promela Manual. http://spinroot.com/spin/Man/promela.html [ac-

cessed 2021-04-17].

[20] Lima, V.—Talhi, C.—Mouheb, D.—Debbabi, M.—Wang, L.—Pourzandi, M.: Formal Verification and Validation of UML 2.0 Sequence Diagrams Using Source and Destination of Messages. Electronic Notes in Theoretical Computer Science, Vol. 254, 2009, pp. 143–160, doi: 10.1016/j.entcs.2009.09.064.

[21] Vidal-Silva, C. L.—Villarroel, R.—Rubio, J.—Johnson, F.—Madariaga, E.—Campos, C.—Carter, L.: An Spin/Promela Application for Model Checking UML Sequence Diagrams. International Journal of Advanced Computer Science and Applications, Vol. 9, 2018, No. 10, doi: 10.14569/IJACSA.2018.091071.

[22] Holzmann, G. J.: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, 1997, No. 5, pp. 279–295, doi: 10.1109/32.588521.

[23] Hlaoui, Y. B.—Younes, A. B.—Ben Ayed, L. J.—Fathalli, M.: From Sequence Diagrams to Event B: A Specification and Verification Approach of Flexible Workflow Applications of Cloud Services Based on Meta-Model Transformation. 2017 IEEE 41st Annual Computer Software and Applications Conference, (COMPSAC), Vol. 2, 2017, pp. 187–192, doi: 10.1109/COMPSAC.2017.135.

[24] Russo, A. G.: Modeling in Event-B – System and Software Engineering by Jean-Raymond Abrial. ACM SIGSOFT Software Engineering Notes, Vol. 36, 2011, No. 2, pp. 38–39, doi: 10.1145/1943371.1943378.

[25] Messaoudi, N.—Chaoui, A.—Bettaz, M.: A Technique to Validate Automatic Generation of Büchi Automata from UML 2 Sequence Diagrams Based on Multi Layer Transformations. International Journal of Computational Vision and Robotics, Vol. 9, 2019, No. 2, pp. 172–191, doi: 10.1504/IJCVR.2019.098799.

[26] Li, Y.—Turrini, A.—Chen, Y. F.—Zhang, L.: Learning Büchi Automata and Its Applications. In: Bowen, J. P., Liu, Z., Zhang, Z. (Eds.): Engineering Trustworthy Software Systems: 4th International School (SETSS 2018). Springer, Cham, Lecture Notes in Computer Science, Vol. 11430, 2018, pp. 38–98, doi: 10.1007/978-3-030-17601-3_2.

[27] Chabbat, N.—Saidouni, D. E.—Boukharrou, R.—Ghanemi, S.: Formal Verification of UML MARTE Specifications Based on a True Concurrency Real Time Model. Computing and Informatics, Vol. 39, 2020, No. 5, pp. 1022–1060, doi: 10.31577/cai_2020_5_1022.

[28] Ng, M. Y.—Butler, M.: Towards Formalizing UML State Diagrams in CSP. First International Conference on Software Engineering and Formal Methods, 2003, pp. 138–147, doi: 10.1109/SEFM.2003.1236215.

[29] Xu, D.—Philbert, N.—Liu, Z.—Liu, W.: Towards Formalizing UML Activity Diagrams in CSP. Vol. 2, 2008, pp. 450–453, doi: 10.1109/ISCSCT.2008.379.

[30] Bisztray, D.—Ehrig, K.—Heckel, R.: Case Study: UML to CSP Transformation. ResearchGate, 2007.

[31] Varró, D.—Asztalos, M.—Bisztray, D.—Boronat, A.—Dang, D. H.—Geiss, R.—Greenyer, J.—Van Gorp, P.—Kniemeyer, O.—Narayanan, A.—Rencis, E.—Weinell, E.: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In: Schürr, A., Nagl, M.,

Zündorf, A. (Eds.): Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5088, 2007, pp. 540–565, doi: 10.1007/978-3-540-89020-1_36.

[32] Elmansouri, R.—Hamrouche, H.—Chaoui, A.: From UML Activity Diagrams to CSP Expressions: A Graph Transformation Approach Using Atom3 Tool. International Journal of Computer Science Issues (IJCSI), Vol. 8, 2011, No. 2, pp. 368–374.

[33] Elmansouri, R.—Meghzili, S.—Chaoui, A.: A UML 2.0 Activity Diagrams/CSP Integrated Approach for Modeling and Verification of Software Systems. Computer Science, Vol. 22, 2021, No. 2, pp. 209–233, doi: 10.7494/csci.2021.22.2.3478.

[34] AToMPM: A Tool for Multi-Paradigm Modelling. `https://atompm.github.io/` [accessed 2021-05-25].

[35] GROOVE: Graphs for Object-Oriented Verification. `https://groove.ewi.utwente.nl/about` [accessed 2021-05-25].

[36] Kerkouche, E.—Khalfaoui, K.—Chaoui, A.: A Rewriting Logic-Based Semantics and Analysis of UML Activity Diagrams: A Graph Transformation Approach. International Journal of Computer Aided Engineering and Technology, Vol. 12, 2020, No. 2, pp. 237–262, doi: 10.1504/IJCAET.2020.10026291.

[37] De Busser, J.: Model Checking AToMPM Transformation Systems with Groove. Technical Report. MSDL Lab, McGill University, Canada, 2018.

[38] Meghzili, S.—Chaoui, A.—Strecker, M.—Kerkouche, E.: On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL. 2017 IEEE International Conference on Information Reuse and Integration (IRI), 2017, pp. 419–426, doi: 10.1109/IRI.2017.63.

[39] da Costa Cavalheiro, S. A.—Foss, L.—Ribeiro, L.: Theorem Proving Graph Grammars with Attributes and Negative Application Conditions. Theoretical Computer Science, Vol. 686, 2017, pp. 25–77, doi: 10.1016/j.tcs.2017.04.010.

[40] Cuadrado, J. S.—Guerra, E.—de Lara, J.: Static Analysis of Model Transformations. IEEE Transactions on Software Engineering, Vol. 43, 2016, No. 9, pp. 868–897, doi: 10.1109/TSE.2016.2635137.

[41] Meghzili, S.—Chaoui, A.—Strecker, M.—Kerkouche, E.: Verification of Model Transformations Using Isabelle/HOL and Scala. Information Systems Frontiers, Vol. 21, 2019, No. 1, pp. 45–65, doi: 10.1007/s10796-018-9860-9.

[42] Nipkow, T.—Wenzel, M.—Paulson, L. C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2283, 2002, doi: 10.1007/3-540-45949-9.

[43] Wenzel, M.: Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit. Electronic Notes in Theoretical Computer Science, Vol. 285, 2012, pp. 101–114, doi: 10.1016/j.entcs.2012.06.009.

[44] Gardner, W. B.: CSP++: How Faithful to CSPm? In: Broenink, J. F., Roebbers, H. W., Sunter, J. P., Welch, P. H., Wood, D. C. (Eds.): Communicating Process Architectures 2005. IOS Press, Concurrent Systems Engineering Series, Vol. 63, 2005, pp. 129–146.

[45] CSPM. `https://cocotec.io/fdr/manual/cspm.html` [accessed 2021-04-17].

[46] Sun, J.—Liu, Y.—Dong, J. S.—Chen, C.: Integrating Specification and Programs for System Modeling and Verification. 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 2009, pp. 127–135, doi: 10.1109/TASE.2009.32.

[47] Shi, L.—Liu, Y.—Sun, J.—Dong, J. S.—Carvalho, G.: An Analytical and Experimental Comparison of CSP Extensions and Tools. In: Aoki, T., Taguchi, K. (Eds.): International Conference on Formal Engineering Methods (ICFEM 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7635, 2012, pp. 381–397, doi: 10.1007/978-3-642-34281-3_27.

[48] Poole, J. D.: Model-Driven Architecture: Vision, Standards and Emerging Technologies. Workshop on Metamodeling and Adaptive Object Models (ECOOP 2001), 2001.

[49] Kleppe, A. G.—Warmer, J.—Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., 2003.

[50] Ehrig, H.—Engels, G.—Kreowski, H. J.—Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation – Volume 2: Applications, Languages, and Tools. World Scientific Publishing Company, 1999, doi: 10.1142/4180.

[51] Kahani, N.—Bagherzadeh, M.—Cordy, J. R.—Dingel, J.—Varró, D.: Survey and Classification of Model Transformation Tools. Software and Systems Modeling, Vol. 18, 2019, No. 4, pp. 2361–2397, doi: 10.1007/s10270-018-0665-6.

[52] Wieber, M.—Anjorin, A.—Schürr, A.: On the Usage of TGGs for Automated Model Transformation Testing. In: Di Ruscio, D., Varró, D. (Eds.): Theory and Practice of Model Transformations (ICMT 2014). Springer, Cham, Lecture Notes in Computer Science, Vol. 8568, 2014, pp. 1–16, doi: 10.1007/978-3-319-08789-4_1.

[53] Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J. L., Nagl, M., Böhlen, B. (Eds.): Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3062, 2003, pp. 446–453, doi: 10.1007/978-3-540-25959-6_35.

[54] Balasubramanian, D.—Narayanan, A.—van Buskirk, C. P.—Karsai, G.: The Graph Rewriting and Transformation Language: GReAT. Electronic Communications of the EASST, Vol. 1, 2006, doi: 10.14279/tuj.eceasst.1.89.

[55] Lutz, M.: Programming Python. Edition 3rd. O'Reilly Media, Inc., 2006.

[56] Roscoe, A. W.: The Theory and Practice of Concurrency. Edition 3rd. Prentice Hall PTR, 1997.

[57] The Coq Development Team: The Coq Proof Assistant Reference Manual. 2004, `http://coq.inria.fr` (Version 8.0).

Houda Hamrouche is Assistant Professor in the Department of Computer Science, 20 Aout 1955 University of Skikda, Algeria. She received her Master degree in computer science in 2011 from the same university. Her research field is UML, formal methods, and distributed systems.

Allaoua Chaoui is Full Professor of computer science with the Department of Computer Science and Its Applications, Faculty of New Technologies of Information and Communication, University Abdelhamid Mehri-Constantine 2, Algeria. He received his Master degree in computer science in 1992 from the University of Constantine (in cooperation with the University of Glasgow, Scotland) and his Ph.D. degree in 1998 from the University of Constantine (in cooperation with the CEDRIC Laboratory of CNAM in Paris, France). He has served as Associate Professor in the Philadelphia University in Jordan for five years and in the University Mentoury Constantine for many years. During his career, he has designed and taught courses in software engineering and formal methods. He has published many articles in international journals and conferences. He supervised many Master and Ph.D. students. His research interests include mobile computing, formal specification and verification of distributed systems, and graph transformation systems and their correctness.

Smaine Mazouzi received his M.Sc. and Ph.D. degrees in computer science from the University of Constantine, respectively in 1996, and 2008. He is Professor at 20 Aout 1955 University of Skikda and the Head of AI and DAI team at the LICUS Laboratory (Laboratoire d'Informatique et de Communication de l'Université de Skikda), at the same university. His fields of interest are pattern recognition, machine vision, distributed systems, and computer security. His current research concerns using distributed and complex systems modeled as multi-agent systems in image understanding and intrusion detection. He is the member of several national and international research projects in computer vision and computer security.