Transforming of the Sequence Diagram into Time-Automata Network

S. Duangmalai, and C. Dechsupa

Abstract—**Formal verification using a model checking approach is a process for proving undesirable properties in designed models. The model checking procedure for the sequence diagram is cumbersome because the transformation of the sequence diagram into a formal model requires meticulous mapping rules and methods that must yield corresponding behaviors. This paper proposes the transformation of the sequence diagram into a time automate named UPPAAL. The obtained time automata model can be used to verify deadlock, undesirable properties, and the correctness of message ordering. The transformation rules and framework were experimented with case studies. The results show that the proposed transformation rules can be applied and map the sequence diagram into a UPPAL structure correctly.**

Index Terms—**Formal verification, UML sequence diagram, Time automata, Software engineering**

I. Introduction

FORMAL verification is the proving process that helps the system analyst localize undesirable properties in a software model. It directly correlates with the quality of the obtained software. Sequence diagram (SD) is a typical tool used in the software design process. The SD may contain mistakes or undesirable properties [1]: deadlock, livelock, and so on. Especially, the SD contains incorrect message ordering in which a receiver does not get a required message on time or it receives an incorrect message sequence that may affect a system's processing. An interaction between objects in the SD represents the messages between objects based on the method, parameter, and time. The system designers can verify the SD to find errors and check whether it conforms to system requirements or not by using model checking approaches [2]. It can also measure the efficiency and effectiveness of the SD as well.

An error finding of the SD starts with transforming the SD elements into the formal model written in a specific formal language [3]. Next, the formal model will be imported into the verification tool to find errors. The formal model abstraction may be quite a difficult process for the designers if they are inexperienced in the formal language. Therefore, this paper proposes the transformation rules and method for mapping the sequence diagram into the formal model described in a network time automaton. The SD elements must be transformed into a formal model by using the framework in which the transformation rules have to produce the target formal model behaviors that conform to the behaviors of the origin SD. The framework advocates transparent mapping, in which the modelers can transform the SD automatically without grammatical and lexical UPPAAL background and the target models can be verified by using the UPPAAL environment [4]. The typical properties of a model, such as the message ordering, deadlock, livelock, and the specific system requirements can also be verified by expressing the properties in computation tree logic (CTL) [5] provided by the UPPAAL environment. The framework will help the software designers improve the model beforehand. It reduces mistakes in software models, decreases development costs, and shortens development time.

The organization of this paper is as follows; Section II describes the background of the SD and UPPAAL. Section III discusses the related works, and Section IV details the research methodology and experiments. Sections V and VI are implementation, validation, and conclusion respectively.

Manuscript received Feb 28, 2023; revised Mar 28, 2023.

S. Duangmalai is a postgraduate student of Computer Science Department, College of computing, Khon Kaen University, Muang Khon Kaen, 40002, Thailand. (e-mail: sumate.d@kkumail.com).

C. Dechsupa is an assistant professor at department of computer science, College of computing, Khon Kaen University, Khon kaen,40002, Thailand (to provide phone: (+66) 043-009700, 50525; e-mail: chanode@kku.ac.th).

II. Background

A. *UML Sequence Diagram [6]*

A sequence diagram or SD is one of the Unified Modeling Language (UML) diagrams, representing the message interactions between objects. The objects in the SD show lifelines, and activated bars are used to represent activities occurring from classes or objects. Whereas the message symbol that links between objects shows the message interactions. The message interpretation typically starts from the object on the top left-hand side and moves to the object on the right-hand side depending on the direction. The core elements and an example of SD are shown in Fig. 1. The model comprises the objects *Customer* and *Order* where the parameter v1 passes a value from the object customer to the object order by using the method *Request*. The message *Return* occurs after receiving and processing the object *Order*. However, the message *Request* and *Return* relies on the evaluation of the condition in the loop fragment if the variable *v1* is true only. As the mentioned diagram indicates, the messages ordering of the object *Customer* must be *Request* → *Return*; or *t* and *t+n*, where *t* is a local clock as the message *Request* is sent and *t+n* is the message *Return* is received by the object *Customer*.

B. *UPPAAL [4]*

UPPAAL was invented by Uppsala University for creating and verifying real systems that are modeled as timed automata networks. It provides verification environments that come along with description language, simulation, and verification tools. The formal system can be modeled as a timed automata network with clock and data variables. The core elements of UPPAAL are detailed in Fig. 2.

An initial state is portrayed in a double-line cycle while a single-line cycle shows the ordinary system state called *Location*. The location name of UPPAAL model must be unique, and it will be determined in part of the CTL to trace the system working at verification stage. A state transition uses a directed edge to connect between the locations. For each state transition may or may not rely on a transition condition with time and data constraint. For instance, the guard condition *v1==true* in Fig.2 is that the state transition occurs if the variable *v1* is true, and value of the variable *c1* is updated by the expression c*1: =c1+1*. The modelers can also describe the state transition in terms of the communication channel and synchronization as well.

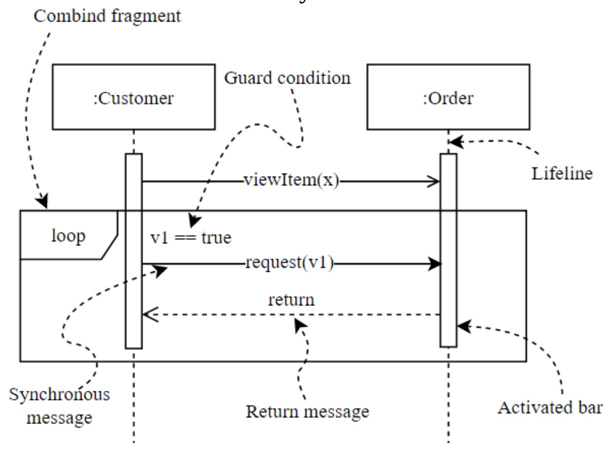


Fig. 1. Core elements and the SD represent the interactions between two objects (*Customer* and *Order*). The model also contains the loop fragment in which the synchronous messages proceeded depend on when the Boolean condition *v1* is true.

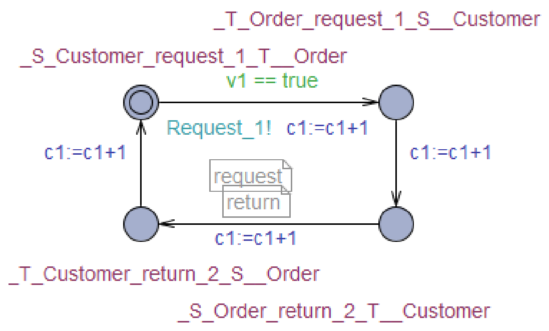

_T_Order_request_1_S__Customer
_S_Customer_request_1_T__Order
v1 == true
Request_1! c1:=c1+1
c1:=c1+1
c1:=c1+1
request
return
c1:=c1+1
_T_Customer_return_2_S__Order
_S_Order_return_2_T__Customer

Fig. 2. An example of UPPAAL construct representing the cycle flow.

III. Related Work

Andrade et al. [7] provided an approach to model and analyze real-time and embedded systems. The authors used a time Petri net with energy constraints or ETPN as a verification tool. Time and energy constraints of the real-time system models are analyzed during the early design stages by transforming the SD into an ETPN construct. The target constructs convey the corresponding behaviors covering with both time and energy constraints. They are used for evaluation of the best path, the worst path, and the energy of the models. However, model transformation is still a manual process.

Chen et al. [8] provided a method based on events to increase the accuracy of SD. The authors used propositional projection temporal logic or PPTL as an automaton to describe the formal model of SD, and desired properties of the model are verified. They also provided an efficient mechanism for checking the SD by implementing a model based on event-deterministic finite automata (ETDFA). The SD properties written in PPTL and implemented with a model checker are used to validate the properties.

Cunha et.al [9] provided a method for transforming the SD to Petri nets and verified the deadlock, reachability, safety, and liveness properties. This technique is a model checking technique that the Petri net model can be designed and verified by using a tool named “FOREVER”. A Symbolic Model Verifier: SMV is a tool for modeling the SD in Petri nets, and it is used for tracing the model properties that are expressed in the computation tree logic.

V.Lima et.al [10] proposed the verification and validation techniques using the SPIN model checker [11]. It is used to trace the execution states of UML sequence diagrams. The authors provided PROMELA structures in which the source message and destination message in the sequence diagram are expressed in LTL. These techniques are applied in our work by mapping the source message and destination message into the UPPAAL process template.

IV. Methodology

An overview of the SD verification is shown in Fig. 3. The transformation process consists of 4 steps: 1) extract the SD elements from an XML file of SD that is designed and exported from Draw.io [12]. 2) All the SD elements derived from the first step are mapped into UPPAAL constructs by using the transformation rules, and 3) the modelers can export the UPPAAL construct and refine the time constraint, and 4) verify the derived UPPAAL construct by using the UPPAAL verification tool.

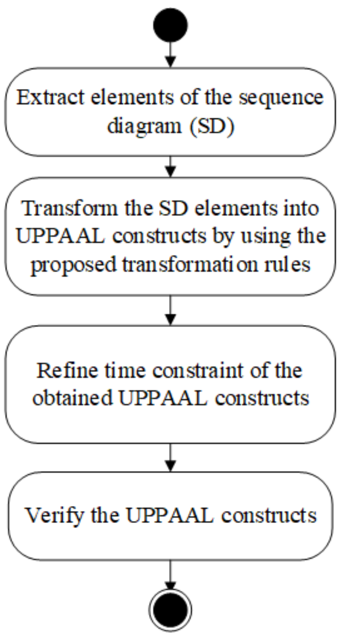

Extract elements of the sequence diagram (SD)
Transform the SD elements into UPPAAL constructs by using the proposed transformation rules
Refine time constraint of the obtained UPPAAL constructs
Verify the UPPAAL constructs

Fig. 3. The SD verification process.

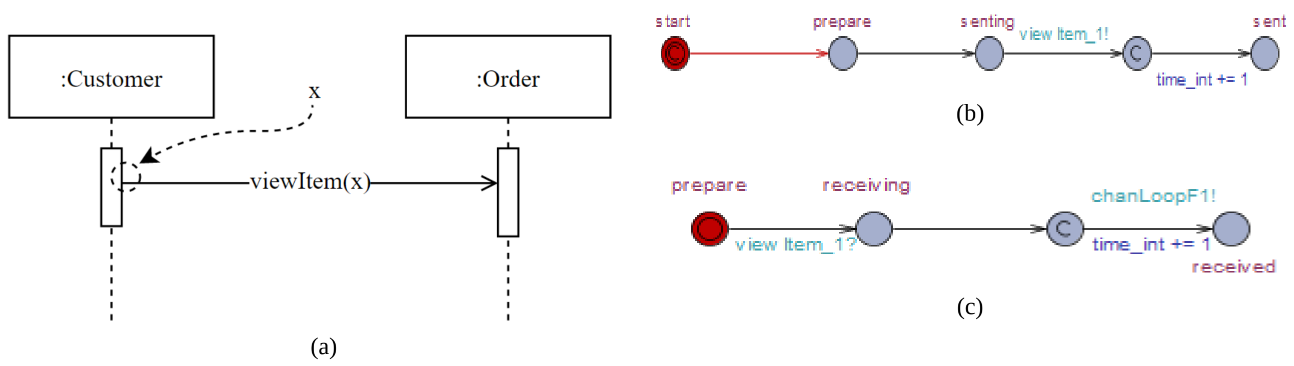


Fig. 4. Transformation rules of the message, (a) the SD, (b) UPPAAL initial message, (c) UPPAAL ordinary message.

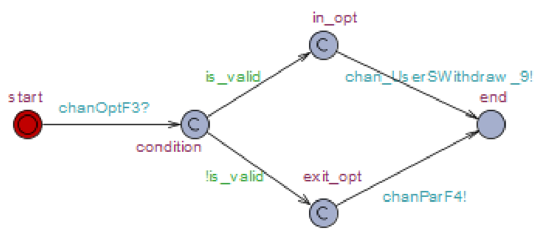


Fig. 5. Transformation rules of the alternative and optional combined fragment.

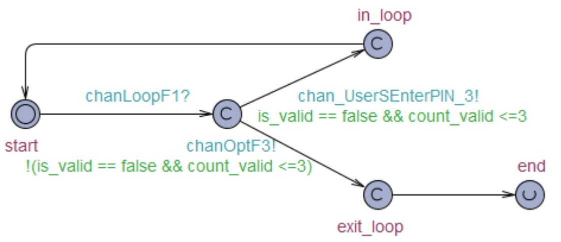

in_loop
chanLoopF1?
chan_UserSEnterPIN_3!
is_valid == false && count_valid <=3
start
chanOptF3!
!(is_valid == false && count_valid <=3)
end
exit_loop

Fig. 6. Transformation rules of the loop combined fragment.

We defined the formal definitions of the relevant models to show the relationships of their elements as follows.

Definition 1 sequence diagram: an ordinary sequence diagram is a thirteen- tuple, *SD* = (*O*, *L*, *A*, *N*, *FG*, *M*, *Z*, *FQ*, *F*, *P*, *FA*, *FM*, *FT*) where

O is a set of objects, where *O.name* is the object name.

L is a set of *liveliness*.

A is a set of activated bars; where *A.id* is the identifier number, and *A.seq* indicates the sequence of an activated bar on the liveline *L*.

N is a set of combined fragments.

FG is a function identifying a fragment type; *FG*: *N* ⟶ (*Alt*, *Loop*, *Opt*, *Par*, …)

M is a set of messages.

Z is a set of methods; where *Z.name* is the method name and *Z.par* is the list of parameters.

FQ is a function that determines the message ordering of the message $m \in M$ on the liveline *L*; *FQ*: (*L*, *M*) ⟶ ℕ.

F is a set of messages that attach on the message, where *F.LabelName* is the message name or the method name, and *F.Type* = {*Label*, *Method*}.

P is a set of parameters.

FA is a mapping function used to indicate the liveline host or the object of an activated bar; *FA*: *A* ⟶ *L* or *FA*: *A* ⟶ *O*.

FM is a mapping function used for determining a message type; *FM*: *M* ⟶ (*SynMSG*, *Asing*, *ReturnMSG*)

FT is a mapping function for identifying a message type; *FT*: *M*⟶*F*.

Definition 2 *Combined fragment*: A sub-process in a combined fragment is a sub-process of parent process, which the SD is hierarchical structure, *FM* = (*SSD*, *FR*) where

SSD is a set of ordinary sequence diagrams.

FR is a mapping function used to indicate the combined fragment in the SD. In short, *FM* is a special SD in which the message *mi* ∈ *M* such that *FR*(*mi*) = *oi* where *oi* ∈ *SSD*

Definition 3 *UPPAAL construct*: an UPPAAL construct is an eight- tuple, *UPP* = (*S*, *T*, *L*, *I*, *V*, *C*, *G*, *Pc*) where

S is a set of states or locations.

T is a set of transitions or edges.

L is a set of labels.

I is an initial state; *I* ⊆ *S*.

V is a set of variables.

C is a set of channels.

G is a set of guards, *g* ∈ *G* is a conditional expression which the used variables refer to the variable *v* ∈ *V*.

Pc is a set of processes.

The SD transformation rules.

We proposed the core transformation rules as follows:

Rule 1: For each event occurring with an object of the SD is mapped to be the part of process name. The process comes from the event of sending or receiving a message. The naming convention of process is:
"*ObjectName_Direction_MessageLabel_MessageOrder*", where *Direction* is a source or destination of message that is determined as *S* and *D* respectively. The *MessageLabel* may be a label name or method name while *MessageOrder* is message number. The process named "*UserSInsertCard_1*" in Fig. 4 (b) represents the event that derived from the event marked "*x*" in Fig. 4 (a).

Rule 2: For each synchronous message and asynchronous message *m where m* ∈ *M* on an activated bar and *FM: m*⟶{*SynMSG*, *AsynMSG*, *ReturnMSG* }. A set of locations *S1* and *S2* are created to represent a source and destination message. It can be said that each message in SD is mapped into two sets of locations (set of source and destination UPPAAL process). The source process contains four locations: *prepare*, *sending*, *commit* and *sent*, whereas the destination process consists of *prepare*, *receiving*, *commit* and *received*. Fig. 4 (c) shows an example of synchronous message transformation.

In case of an initial process of SD, the first message on the left top *m* is a message in the SD and *FQ*(*m*) =1, the location

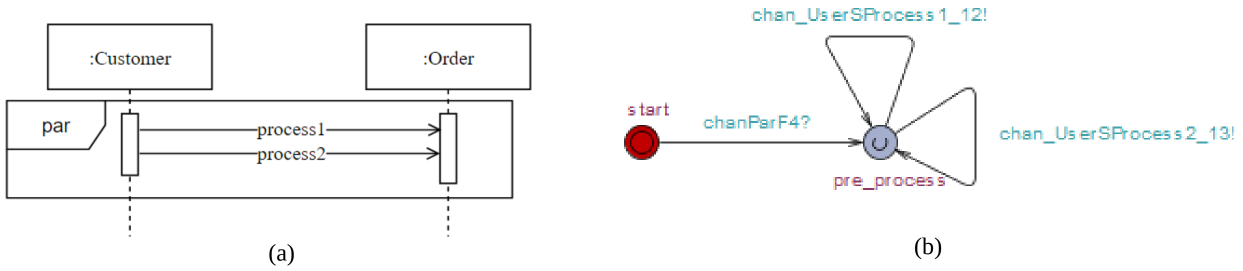


Fig. 7. Transformation rules of the parallel combined fragment.

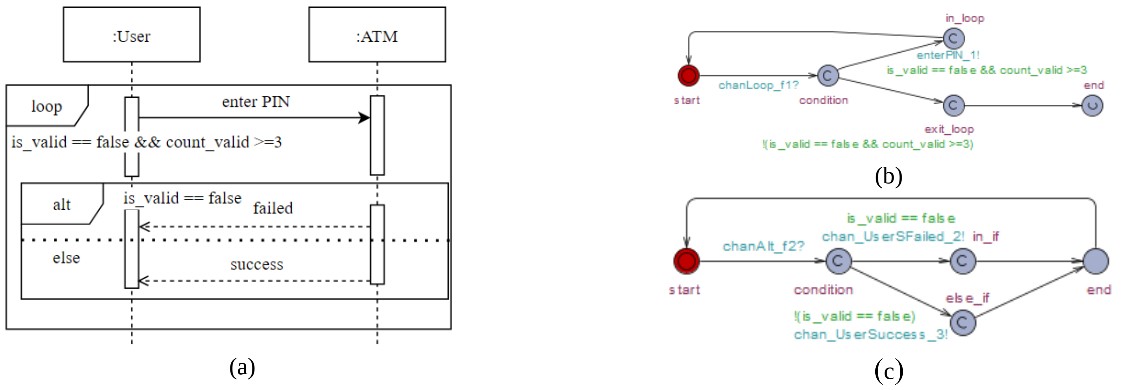


Fig. 8. An example of transformation of the hierarchical combined fragments.

si ∈ *S* is created to be an initial location with the label name *start*, which is prefixed at of the process *pc* ∈ *Pc*.

Rule 3: For each variable in SD, including the parameters of method and variables of a guard condition. It is mapped into a variable *v* ∈ *V* directly.

Rule 4: Guard condition on each message or on a combined fragment is mapped in a guard condition *g* ∈ *G*.

Rule 5: As the transformation rule no. 2, the communication between objects or UPPAL processes needs a channel for data synchronization. The channel name is generated from the label or *Method name*, and *_MessageOrder*. Fig. 2 shows the channel name *"Request_1*" derived from the SD message *Request(v1)* in Fig. 1.

Rule 6: Alternative and optional combined fragment in SD *FG*: *N* ⟶ {*Alt*, *Opt*} closes to the *If-else* pattern that contains two boundaries (If-boundary and else-boundary). The SD elements in each boundary are transformed by using rule no. 1 to 5. Next, the processes that come from both boundaries are connected by an UPPAAL control flow. In short, a combined fragment is mapped into the UPPALL process *pc* ∈ *Pc*, and it is an intermediary synchronizing data between processes by using a channel. The UPPAAL construct of an alternative and optional control flow is shown in Fig. 5, where the location named *condition* conveys the flow direction relied on the guard conditions *is_valid* and *!is_valid* on its outgoing edges, and the process of the If-boundary and Else-boundary are called by using the channels.

Rul 7: For each loop combined fragment of the SD is transformed into an UPPAAL control process. Fig. 6 represents the UPPAAL loop-control process where the location named *condition* conveys the process flows based on a guard condition "*v1==true*", and the edge "*chanLoopF1*" acts as a control loop firing a token back to the location *condition* again. If the guard condition of the loop is evaluated to be true, the control loop process will proceed with the sub-process under the loop boundary via the channel *chan_CustomerSRequest_1*.

Rul 8: If the SD message is in a loop combined fragment, it is mapped using rule no.2, and a reset-edge is added between the last location and the location named *prepare* to produce a token back for the next round. An example of the UPPAAL construct that is obtained from the message in the combined fragment is shown in Fig. 6.

Rul 9: Parallel combined fragment in the SD is transformed into an urgent location. Its input must come from the same channel, next it proceeds all processes by calling the channels of each UPPAAL construct simultaneously. An example of the parallel combined fragment is shown in Fig. 7 (a), and the derived UPPAAL construct of the parallel combined fragment is represented in Fig. 7 (b).

Rule 10: Multiple combined fragments with sub-combined fragment. It indicates that the SD contains fragment occupies a combined fragment hierarchically. All elements of them are transformed into UPPAAL constructs by using all above rules. The channels are created to concatenate the processes within each the fragment boundary and the process outside of the fragment boundary.

Rule 11: Due to the SD without clock determination, a clock of each UPPAAL location is determined as 1 by default. This rule covers the typical location only, while the commit and urgent location does not cover because their clocks are 0 intuitively.

The overview of transformation rules is that the set of SD elements are transformed and grouped to be sub-processes based on the objects in the SD. For each sub-process can also be partitioned as a sub-process once again if the SD contains a combined fragment. The messages and methods between objects are mapped into communication channels and parameters. Although the time constraint cannot be determined in the SD, the transformation generates the local time counters, and their values will be increased by 1. The modelers can adjust the time constraints on each message event in the UPPAAL construct before verification arbitrarily.

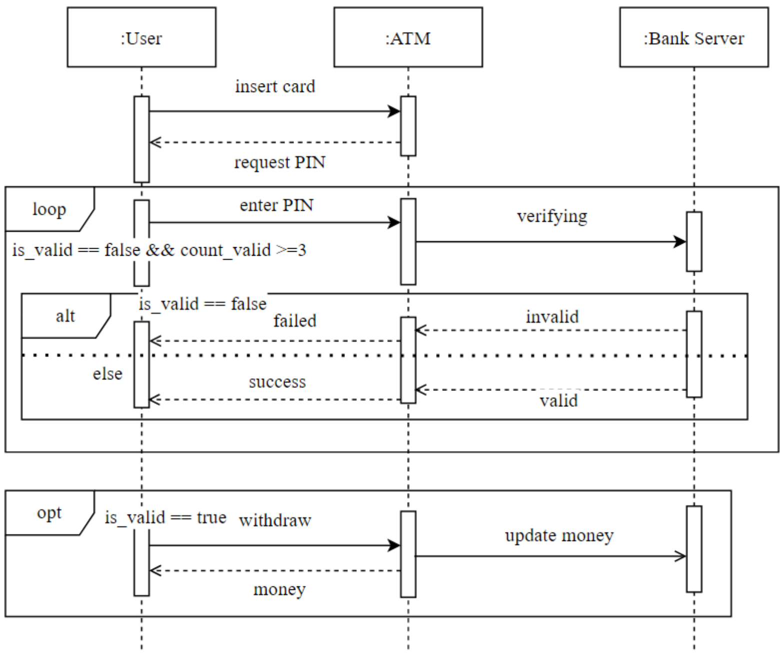


Fig. 9. The ATM sequence diagram applied with the proposed transformation rules.

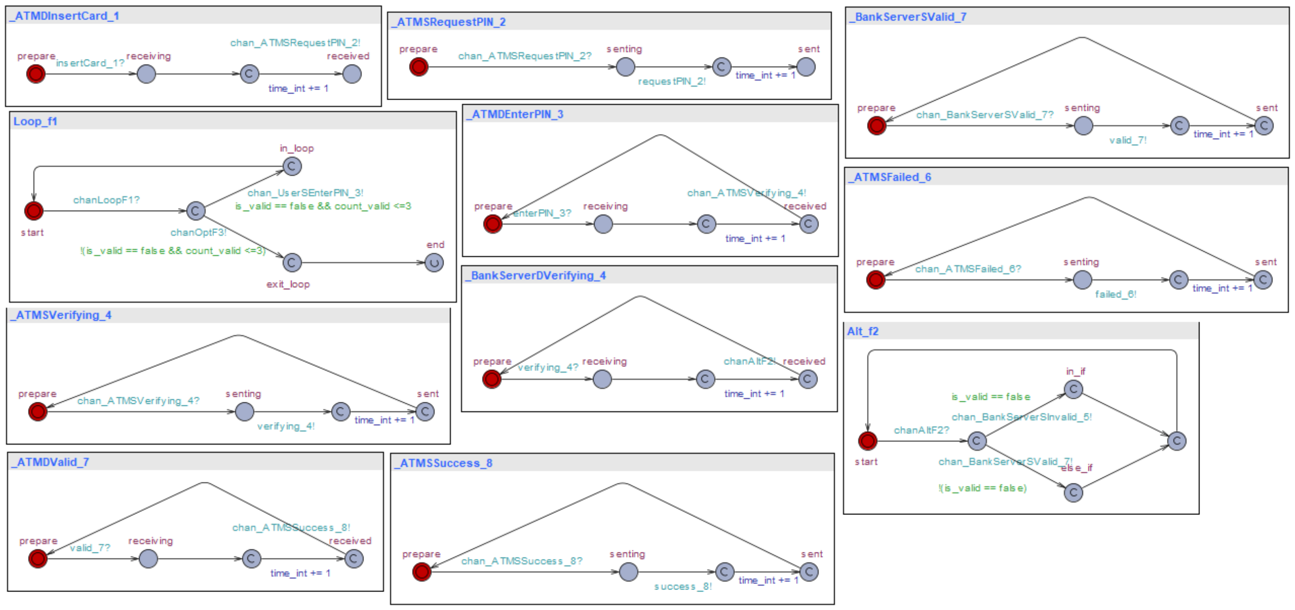
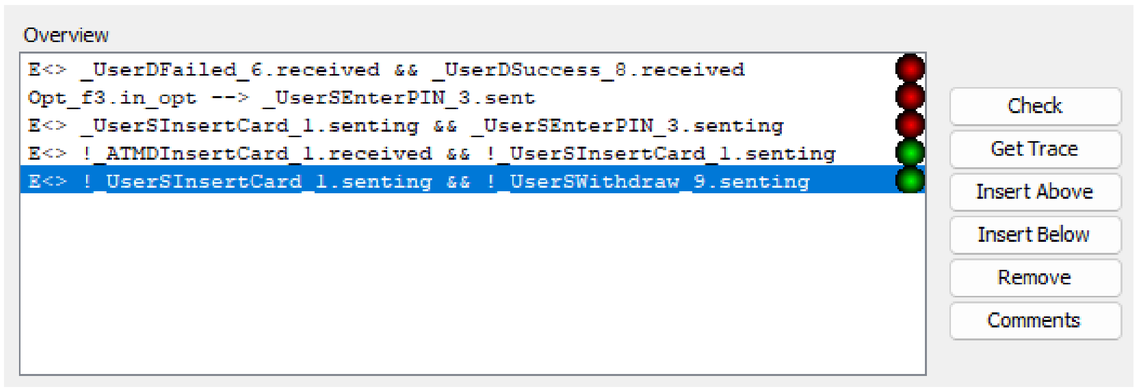


Fig. 10. The excerpt obtained UPPAAL constructs of the ATM sequence diagram in Fig 9.



```
E<> _UserDFailed_6.received && _UserDSuccess_8.received
Opt_f3.in_opt --> _UserSEnterPIN_3.sent
E<> _UserSInsertCard_1.senting && _UserSEnterPIN_3.senting
E<> ! _ATMDInsertCard_1.received && ! _UserSInsertCard_1.senting
E<> ! _UserSInsertCard_1.senting && ! _UserSWithdraw_9.senting
```

Fig. 11. A screenshot of the UPPAAL constructs verified in the UPPAAL tool by using verification mode.

TABLE I
THE EXCERPT RESULTS OF UPPAAL ATM MODEL VERIFICATION IN VERIFICATION MODE OF UPPAAL ENVIRONMENT

No	Requirements	CTL	Results
1	The *user* object cannot send the messages *insertCard* and *withdraw* to the ATM object simultaneously.	E<> !_UserSInsertCard_1.senting && !_UserSWithdraw_9.senting	Satisfied
2	The ATM object will not be enable to receive the message *insertCard* if the object user has not sent it yet.	E<> !_ATMDInsertCard_1.received && !_UserSInsertCard_1.senting	Satisfied
3	The *user* object cannot send the message *insertCard* and *EnterPIN* at the same time.	!E<> _UserSInsertCard_1.senting && _UserSEnterPIN_3.senting	Satisfied
4	Is it possible that when the optional fragment is evaluated to be *true*, the *user* then sends the message *enterPIN* to the ATM object?	Opt_f3.in_opt --> _UserSEnterPIN_3.sent	Unsatisfied
5	The alternative fragment proceeds either *UserFailed* nor *UserDSuccess*.	!E<> _UserDFailed_6.received && _UserDSuccess_8.received	Satisfied

V. IMPLEMENTATION AND VALIDATION

We validated the transformation rules by using a case study to show that the obtained UPPAAL constructs can be verified in UPPAAL verification framework. The SD model of the ATM system shown in Fig. 9 is transformed into the UPPAAL constructs. The SD model comprises three objects (*User*, *ATM* and *Bank_server*), twelve messages and three combined fragments. After applying the proposed transformation rules, we obtained the UPPAAL constructs shown in Fig. 10. The UPPAAL environment is used to create, modify the UPPAAL constructs the model layout. It also is used to refine the clock constraint, including the CTL expression for the properties exploring the specific desirable behaviors of the model such as deadlock, liveness, and soundness properties.

As the UPPAAL constructs shown in Fig. 10, we verified the ATM system in a simulation mode and verification mode of the UPPAAL environment tools. For instance, we simulated the model to trace back and forward steps of a message sending and receiving. Moreover, all variables of the model can be monitored for their values during simulation. Whereas the verification mode is performed based on our CTL expression. For example, we found the unreachable part in the model. The message "success" cannot be reached because the guard condition of the fragment cannot be evaluated to be true. The CTL for checking this event is "E<> _UserDSuccess_8.received", which means that "success" does not satisfied this property. Due to the space limitation, we show excerpt CTL used for verifying the UPPAAL constructs of the ATM system in Table I.

VI. CONCLUSION

After designing the sequence diagram, the designers verify their models to check if the model properties meet the desired properties or not. The diagram may contain complex messages and many combined fragments that are difficult to verify in an ordinary testing technique. We proposed the transformation rules for mapping the sequence diagram or SD into a time-automata network of UPPAAL constructs. The transformation rules cover the SD objects, messages, fragments, variables, including the guard conditions. The derived UPPAAL constructs can be verified in UPPAAL environment in the simulation mode and verification mode. We validated the transformation rules by using a case study of the ATM system. The obtained ATM UPPAAL constructs are explored with the liveness, and specific properties expressed in CTL. From the experiment, we observe that the transformation rules advocate the sequence diagram verification with time constraints correctly. However, the limitation of verification is that the sequence diagram is without time properties, whereas the UPPAAL time-automata needs the time constraints. Thus, after applying the transformation rules the models require an adjustment to the time-constraints.

We will extend the transformation rules handling the other fragments such as critical region, negative, break, weak sequencing, strict sequencing, ignore/consider, assertion, and will develop a SD designer plugin to generate the UPPAAL model automatically.

REFERENCES

[1] Li, X., Liu, Z., & Jifeng, H. "A formal semantics of UML sequence diagram," In: 2004 Australian Software Engineering Conference. Proceedings. IEEE, 2004. p. 168-177.
[2] Clarke, E. M. "Model checking," In: Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17. Springer Berlin Heidelberg, 1997. p. 54-56.
[3] Salomaa, A. "Formal languages,". Academic Press Professional, Inc., 1987.
[4] Behrmann, G., David, A., & Larsen, K. G. "A tutorial on uppaal," Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures, 2004, 200-236.
[5] Hafer, T., & Thomas, W. "Computation Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree," In: ICALP. 1987. p. 269-279.
[6] Object Management Group, "OMG Unified Modeling Language TM (OMG UML) version 2.5," 2015.
[7] Andrade, E., Paulo M., Gustavo C., Bruno N., and Carlos A. "Mapping UML sequence diagram to time petri net for requirement validation of embedded real-time systems with energy constraints." In Proceedings of the 2009 ACM symposium on Applied Computing, 2009, p. 377-381.
[8] Chen, Z., and Duan Z.. "Specification and verification of UML2. 0 sequence diagrams using event deterministic finite automata." In SSIRI-C 2011, 2011, p. 41-46.
[9] Cunha, E., Marcelo C., Herbert R., and Barreto. "Formal verification of UML sequence diagrams in the embedded systems context." In SBESC 2011, 2011, p. 39-45.
[10] Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L. and Pourzandi,. "Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages",. Electronic notes in theoretical computer science, 2009, p.143-160.
[11] Holzmann, G.J., 1997. The model checker SPIN. IEEE Transactions on software engineering, 23(5), p.279-295.
[12] Draw.IO, "About Draw.IO," Accessed: Feb. 14, 2023. [Online]. Available: http://about.draw.io.