A Method for Modeling and Verifying of UML 2.0 Sequence Diagrams using SPIN

Chi Luan Le

Abstract— **This paper proposes a method for modeling and verifying UML 2.0 sequence diagrams using SPIN/PROMELA. The key idea of this method is to generate models that specify behaviors of each object in the given UML 2.0 sequence diagrams. In this paper, I/O automata are used as the models to maintain the interaction among objects. This work also proposes a mechanism to translate these models into PROMELA to use SPIN for checking the correctness of the system. By ensuring software design correctness, several properties can be guaranteed such as safety, stability, and the fact that no vulnerability is left. A support tool for this method is presented and tested with some particular systems to show the accuracy and effectiveness of the proposed method. This approach has promising potential to be applied in practice.**

Tóm tắt— **Bài báo này đề xuất một phương pháp để mô hình hóa và kiểm chứng biểu đồ trình tự UML 2.0 sử dụng SPIN/ PROMELA. Ý tưởng chính của phương pháp là xây dựng các mô hình mô tả hành vi của từng đối tượng trong biểu đồ trình tự UML 2.0. Các mô hình này biểu diễn dưới dạng các ôtômát vào/ra nhằm giữ nguyên tính tương tác giữa các đối tượng. Nghiên cứu đưa ra một kỹ thuật hỗ trợ chuyển đổi các mô hình này thành các đặc tả PROMELA để cung cấp cho bộ công cụ SPIN nhằm kiểm chứng tính đúng đắn của các biểu đồ tuần tự. Bằng cách đảm bảo tính chính xác của thiết kế phần mềm, một số thuộc tính có thể được đảm bảo như an toàn, ổn định và thực tế là không còn lỗ hổng nào. Một công cụ hỗ trợ cho phương pháp đề xuất cũng được cài đặt và thực nghiệm với một số hệ thống điển hình nhằm minh chứng cho tính đúng đắn, hiệu quả và dễ sử dụng. Cách tiếp cận này hứa hẹn sẽ được áp dụng trong thực tế.**

Keywords— **Model Checking; Model Generation; SPIN/PROMELA; I/O Automata; Sequence Diagrams.**

Từ khóa— **Kiểm định mô hình; Tạo lập mô hình; SPIN/PROMELA; ô tô mát vào ra; biểu đồ tuần tự.**

This manuscript is received on August 1, 2019. It is commented on August 11, 2019 and is accepted on August 18, 2019 by the first reviewer. It is commented on August 16, 2019 and is accepted on August 22, 2019 by the second reviewer.

I. INTRODUCTION

Software verification in particular and software quality assurant in general play a significant role in software development process, specially when it is crucial to detect flaws before they can be exploited. The verification process can detect errors in early phase of the software lifecycle. Therefore, it greatly reduces bug-fixing, maintaining cost, and efforts in software quality assurant. Currently, one of the most popular methods for software verification is model checking [3, 2].

A prerequisite for using model checking in software verification is to construct models describing behaviors of the system under checking. However, most of the current researches about model checking generally assume the availability and correctness of these models. This assumption may not always hold in practice due to the lack of documentations, model errors, bug-fixing, etc.

Generating models from UML diagrams and model checking them have been known as a potential solution to deal with the above problems. The unified modeling language has become a standard for modeling software architectures and designs. Currently, many techniques have been proposed for models generation and verification of UML diagrams [1, 6, 7, 8 ,9, 11, 12, 13, 15, 16, 18]. Though, most of these approaches targeting sequence diagrams only use an old version of UML. Some methods are targeting UML 2.0. However, they do not handle all newly introduced combined fragments and nested fragments. They only focus on the basic concepts (loop, conditional, etc). In addition, some methods acquire state-space explosion when performing verification. This leads to the limitation in handling large, complex sequence

diagrams. Moreover, the interaction among objects is understated, while it is an important property of sequence diagrams.

To overcome these drawbacks, the work in [18, 10] proposes a different approach which uses a special I/O automata to describe behaviors of one object in sequence diagrams. However, in [18, 10], the method for using these generated models in model checking is not clearly described. In practice, there is no software verification tool which supports I/O automata.

The main contribution of this work is to provide an efficient mechanism for generation of I/O automata into PROMELA [21] processes. SPIN [20] uses these processes and LTL (Linear Temporal Logic) properties as input to provide verification result. The combination of this method with methods in [18, 10] becomes a complete process for specification and verification of UML 2.0 sequence diagrams [19]. The scope of this paper is that the design of software components represented by UML 2.0 sequence diagrams because this is the most detailed behavioral diagram in UML ones. By employing the proposed method, software is guaranteed to contain no error in both functional and security aspects according to its design.

This paper is organized as follows. Related works are presented in section 2. Section 3 describes some improvements in the translation of combined fragments into automata. Section 4 is about the generation of PROMELA file from I/O automata. Support tool and experimental results are presented in section 5. Finally, we conclude the paper and propose some future works in Section 6.

II. RELATED WORKS

Currently, there are many works which are proposed in verification of UML diagrams. Focusing only about sequence diagrams, we can refer to [5, 7, 9, 10, 12, 18].

In [9], Knapp et. al proposes an approach for translating sequence diagrams into interaction automata and uses SPIN as the model checker. This approach does not support all of UML 2.0 combined fragments. In [7], Duong et. al suggests a method of using regular expressions for model generation. Then, the method verifies the generated models by using assume-guarantee verification [4]. However, these models are nondeterministic and the optimization is complex. A mechanism for translation of sequence diagrams into state-machine diagrams is proposed in [5] by Grønmo et. al to take full advantages of state-machine diagrams in verification. The common drawback of [5, 7, 9] is using only one model to describe behaviors of the whole system. This leads to the state-space explosion problem. The complexity of the model affects the performance of verification tools. Furthermore, in these approaches, objects in sequence diagrams are not explicitly described. Last but not least, the interaction among objects is ambiguous.

A method to directly specify sequence diagrams by PROMELA is described in [12]. In this method, an object in sequence diagrams is transformed into a PROMELA process, their sending and receiving events are represented by PROMELA operators. It may overcome the drawbacks of above approaches. However, the authors do not provide a clear description and do not handle nested fragments. Moreover, due to the lack of models as an intermediary role, this method is not flexible, and cannot be used with other verification tools than SPIN. Furthermore, the result also cannot be reused in other phases of software development process.

The work in [18] introduces a different approach. Each object in sequence diagrams is specified by an I/O automaton. Therefore, the whole system is represented by a set of automata. The interaction among objects is represented by send/receive events in each automaton. Authors of [10] improve this method to support more UML 2.0 and nested fragments. However, the use of generated models in model checking is not provided.

In [6] and [8, 11, 13, 15, 16], the translations into PROMELA for activity diagrams and state machine diagrams are presented, respectively. Those works might be used with the process proposed in this paper when systems are designed by different types of diagrams.

III. TRANSLATION OF SEQUENCE DIAGRAMS INTO EDTFA

This paper uses the approach proposed in [18] and the improved method in [10] to specify sequence diagrams by event deterministic finite automata (EDTFA). The approach supports most of UML 2.0 fragments and nested fragments. The approach also describes the interactions among objects.

Though to accurately describe behaviors of objects in the form of PROMELA process, it is necessary to perform some modification to the algorithm of translation of combined fragments into EDTFA, alt, opt and loop fragments in particular.

From [18], we have the definition of objects in sequence diagrams and event deterministic finite automata.

Definition 1: Objects in Sequence Diagram*: An object is a 6-tuple, O= (E, FG, OP, C, num, frag), where:*

- *E is a finite non-empty set of sending and receiving events of the object, |E| is the number of events in E,*
- *FG is a finite set of fragments of the objects,*
- *OP is a finite set of operands,*
- *C is a finite set of guard conditions,*
- *num is a list representing the serial number of events, from 0 to n,*
- *frag is a function from E to F.*

Definition 2: Event Deterministic Finite Automata*: EDTFA are a 7-tuple, M= (Q, C_M, E_M, Σ, δ, q_0, F), where*

- *Q is a finite set of states,*
- *C_M is a finite set of guard conditions of fragments, a guard condition can be also empty, denoted by ε,*
- *E_M is a finite set of events, $E_M = E_{MI} \cup E_{MO}$, E_{MI} is set of receive events, E_{MO} is set of send event,*
- *Σ is a finite set of symbols, ={(c,e)|c ∈C_M, e ∈E_M},*
- *δ is the transition function, δ:Q×(C_M×E_M)→Q,*
- *q_0 is the starting state, q_0 ∈Q;*
- *F is a finite set of final states, F⊆Q.*

With opt, alt, loop fragments, the modification is described as below. With other fragments, the algorithm remains the same as in [18, 10].

With opt fragment, we add an else condition. A transition from the state right before the beginning of the fragment to the state right after the end of the fragment is added. The condition of this transition is the negation of the condition in opt fragment. Here are the new transition rules.

$$\delta(q_i, (\varepsilon, e_j)) = \begin{cases} q_j \text{ if } j = i+1 \text{ and } (e_{j-1} \in FG \text{ or } e_j \notin FG) \\ \text{no definition, otherwise} \end{cases}$$

$$\delta(q_i, (c, e_j)) = \begin{cases} q_j \text{ if } j = i+1 \text{ and } (e_{j-1} \notin FG \text{ or } e_j \in FG) \\ \text{no definition, otherwise} \end{cases}$$

$$\delta(q_i, (!c, e_j)) = \begin{cases} q_j \text{ if } j = i+|FG|+1 \text{ and } \begin{pmatrix} e_j \notin FG \text{ and} \\ e_{j-1} \in FG \end{pmatrix} \\ \text{no definition, otherwise} \end{cases}$$

Figure 1 describes a simple sequence diagram with one opt fragment and the corresponding EDTFA of object A. The new transition is from state q_1 to q_3 when the condition is not satisfied (if !cond, the run will transit from q_1 to q_3 with symbol c).

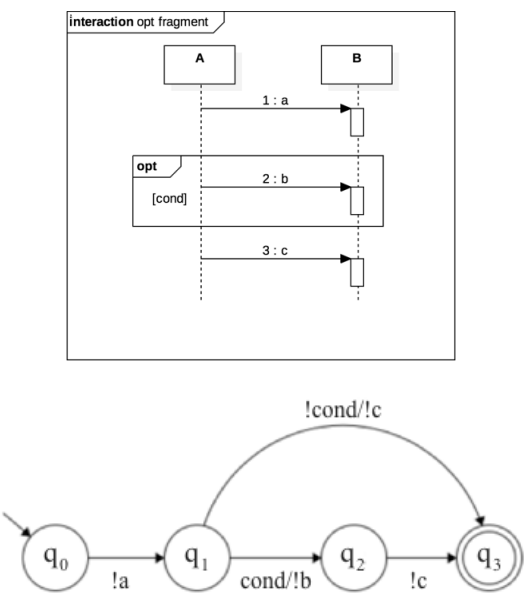

Figure 1. Opt fragment and EDTFA of A

With alt fragment, we add an else condition. A transition from the state right before the beginning of the fragment to the state right after the end of the fragment is added. The condition of this transition is the conjunction of the negation of all operand's conditions in the fragment. Here are the new transition rules.

$$\delta(q_i, (\varepsilon, e_j)) = \begin{cases} q_j \text{ if } j = i+1 \text{ và } \begin{pmatrix} e_j \notin FG \text{ or} \\ (e_{j-1} \in op_k \text{ and } e_j \in op_k) \end{pmatrix} \\ q_j \text{ if } j = i + |op_k| + \cdots + |op_m| + 1 \\ \quad \text{and } e_i \in op_{k-1} \\ \quad \text{and } e_{i+1} \in op_k \\ \text{no definition, otherwise} \end{cases}$$

$$\delta(q_i, (c_k, e_j)) = \begin{cases} q_j \text{ if } j = i+1 \text{ and } k = 1 \\ \quad \text{and } (e_i \notin FG \text{ and } e_{i+1} \in FG) \\ q_j \text{ if } j = i + |op_1| + \cdots + |op_{k-1}| + 1 \\ \quad \text{and } 1 < k \le me_i \\ \quad \text{and } (e_i \notin FG \text{ and } e_{i+1} \in FG) \\ \text{no definition, otherwise} \end{cases}$$

$$\delta(q_i, (!\,c_1 \text{ and } !\,c_2 \text{ and } \ldots \text{and } !\,c_m, e_j)) = \begin{cases} q_j \text{ if } j = i + |op_1| + \cdots + |op_m| + 1 \\ \quad \text{and } (e_i \notin FG \text{ và } e_{i+1} \in FG) \\ \text{no definition, otherwise} \end{cases}$$

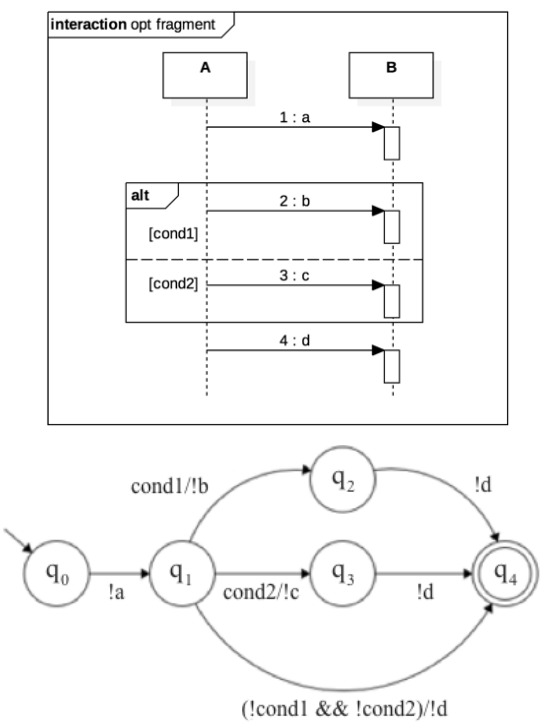

Figure 2. Alt fragment and EDTFA of A

Figure 2 describes a simple sequence diagram with one alt fragment and the corresponding EDTFA of object A. The new transition is from state q_1 to q_4 when all operand's conditions are not satisfied. (if !cond1 and !cond2, the run will transit from q_1 to q_4 with symbol d).

With loop fragment, we add an end condition for the loop. A negation of loop condition is added in any transitions that end the loop.

Figure 3 describes a simple sequence diagram with one loop fragment and the corresponding EDTFA of object A. Transitions ended the loop (from q_1 to q_4 and from q_3 to q_4) have an additional condition which is the negation of loop condition.

Here are the new transition rules.

$$\delta(q_i, (\varepsilon, e_j)) = \begin{cases} q_j \text{ if } j = i+1 \text{ and } (e_{j-1} \in FG \text{ xor } e_j \notin FG) \\ q_j \text{ if } j = i + |FG| + 1 \le n \text{ and } e_{j-1} \notin FG \\ \quad \text{and } e_j \in FG \\ \text{no definition, otherwise} \end{cases}$$

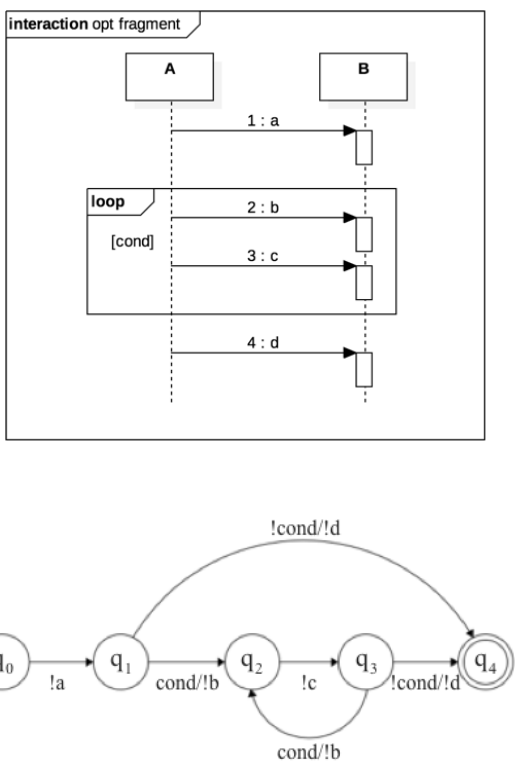

Figure 3. Loop fragment and EDTFA of A

$$\delta(q_i, (c, e_j)) = \begin{cases} q_j \text{ if } j = i + 1 \text{ and } (e_{j-1} \notin FG \text{ and } e_j \in FG) \\ q_j \text{ if } j = i - |loop| + 1 \leq n \text{ and } e_{j-1} \notin FG \\ \qquad \text{and } e_j \in FG \\ \text{no definition, otherwise} \end{cases}$$

$$\delta(q_i, (!\, c, e_j)) = \begin{cases} q_j \text{ if } j = i + |FG| + 1 \leq n \text{ and } e_{j-1} \notin FG \\ \qquad \text{and } e_j \in FG \\ q_j \text{ if } j = i + 1 \text{ và } e_{j-1} \in FG \text{ and } e_j \notin FG \\ \text{no definition, otherwise} \end{cases}$$

After analyzing and extracting data from sequence diagrams and performing translation, we have a set of EDTFAs where each EDTFA specifically describes behaviors of each corresponding object.

IV. AUTOMATIC IMPLEMENTING EDTFA IN PROMELA/SPIN

Due to the lack of verification tool which supports EDTFA, we need to represent these EDTFAs using an input language of any exising tool. We have chosen the SPIN tool for some reasons. SPIN is widely used in both academia and industry as a software verification system. Moreover, its input language, PROMELA, is rather a specification language whose syntax is similar to EDTFA (Table 1), which requires less effort and cost in the generation. After representing EDTFA with PROMELA, SPIN uses the outcome and LTL properties to provide verification results. By this approach, the proposed method can check safety, liveness and other properties that supporting by SPIN.

TABLE 1. MAPPING OF EDTFA ELEMENTS INTO PROMELA

EDTFA elements	PROMELA elements
Automata	Process
States	Label and if block
Symbol	Message
Send/receive events	Send/receive operations

The process of generating PROMELA files from automata includes two steps. In the first step, each automaton will be translated into a PROMELA process (Algorithm 1). Then, in the second step, these processes are combined. Finally, the definitions of symbols and conditions are added into a complete PROMELA file with extension PML (Algorithm 2).

Algorithm 1: Generate PROMELA process from an EDTFA

Input: EDTFA named "obj", M = <Q, C_M, E_{MI}, E_{MO}, Σ, δ, q_0, F>

Output: A PROMELA process describe this automaton

1. let L_0, L_1, ..., L_N is N list of 4-tuple <target state, condition, event, type>, L_i contains all the transition from state Q_i
2. for each transition <q_i, (c, e), q_j> ∈ δ do
3. if e ∈ E_{MI} then
4. add <j, c, e, "receive"> to list L_i
5. end if
6. if e ∈ E_{MO} then
7. add <j, c, e, "send"> to list L_i
8. end if
9. end for
10. let s = "proc" + name + "() {"
11. for each list L_i
12. add new line "qi:" to s
13. add new line "if::" to s
14. if q_i ∈F with stop condition c then
15. add new line with format "::(c) -> goto final"
16. end if
17. for each element <j, c, e, type> L_i
18. if type = send then
19. if c is not empty then
20. add new line with format "::(c) -> msg ! e, goto q_j" to s
21. else add new line with format ":: msg ! e, goto q_j"
22. end if
23. else if type = receive then
24. add new line with format ":: msg ? e, goto q_j"
25. end if
26. add new line "fi"
27. end for
28. add "final: skip" to s
29. add "}" to s
30. return s

The translation of an EDTFA to PROMELA process follows by Table 1 and below rules.

- Each state is transformed to a label. The transitions between states are controlled by goto statement.
- Each send event can be simulated to a message sent by a channel. Each receive event can be simulated to a message received by a channel.
- If block is used in each label to simulate the branch with conditions of this state.

Algorithm 1 will generate a PROMELA process block corresponding to an EDTFA. Firstly, with each state, the algorithm creates a list which contains all transitions that are received or sent by this state (line 1 – 9). Then, the algorithm starts writing a PROMELA process from line 10. With each state and its list created before, the algorithm creates a new label and an if block (line 12 – 13). If this state is a final state, the algorithm adds an operator to go to final label (line 14 – 15). With each transition in the current list, if the transition is "send", the algorithm sends the symbol to the channel and goes to destination state (line 18 – 21). If the transition is "receive", the algorithm reads the symbol from the channel and goes to destination state (line 23 – 24). After proceeding all transitions, the algorithm closes the if block (line 26) and continues with another state. In the end, the algorithm adds a dummy label final to represent the finish of automaton (line 28). Finally, the algorithm closes the process (line 29) and returns a string which describes a PROMELA process.

Algorithm 2: Generate PROMELA of a model described by a set of EDTFAs

Input: A set of EDTFAs which is translated from a sequence diagram, A = {M_1, M_2, …, M_k}

Output: File with PML extensions represent a sequence diagram as PROMELA.

1. let s = "chan msg = [1000] of {mtype};"
2. add new line "mtype = {" to s
3. for each event e ∈ $\bigcup E_{MI_i} \cup E_{MO_i}$
4. add e to current line with delimiter ', '
5. end for
6. add "}" to current line
7. for each condition c ∈ $\bigcup C_{M_i}$
8. define a new variable describing condition c
9. end for
10. for each automaton M_i
11. use algorithm 1 to generate PROMELA process of M_i and add to s
12. end for
13. add new line "init {" to s
14. for each automaton M_i
15. add "run proc<M_i>();"
16. end for
17. add "}"
18. write s to file

Algorithm 2 will create a completed PROMELA file from a set of EDTFAs. Firstly, the algorithm defines a channel which is used for exchanging messages between processes (line 1). Then, the algorithm adds the definitions of symbols (line 2 – 6) and conditions in all the automata (line 7 – 9). Then, the algorithm adds the processes which describe these automata by using Algorithm 1 (line 10 – 12). Finally, the algorithm creates the *init* block to automatically run every process from start (line 13 – 17).

By using Algorithm 2 for the set of EDTFAs, we have a PML file simulated behaviors of sequence diagrams, included the interaction among objects. However, in many cases, this file cannot be directly used by SPIN for some reasons. PROMELA specification does not completely describe system behaviors (for example, missing parameters of events). Syntax errors are also inevitable in the generation process, especially in the definition of symbols and conditions. Furthermore, each type of properties requires a different modification in PROMELA file to monitor variables and events. Therefore, after receiving this PML file, it is necessary to perform an additional step to review and implement required changes before using this file in SPIN.

Currently, the correctness of this method is not theoretical proven but by observation through expertise using with some particular systems. With the comparison between expected result and the outcome of this method, we can confirm that it provides accurate results. Section 5 will present some real systems and the corresponding results when applying these systems with our method.

V. EXPERIMENTAL RESULTS

A tool has been built using JAVA to support the proposed method. Its architecture is presented in Figure 4.

The input of this tool is an XML file that describes a sequence diagram. There is a component which handles the analysis of this sequence diagram and extracts to corresponding objects. With each object, this tool generates a corresponding EDTFA which describes its behaviors. From these EDTFAs, a PROMELA file is generated by using the proposed algorithm. SPIN uses this file as the input. Then, it provides verification results of properties described by LTL.

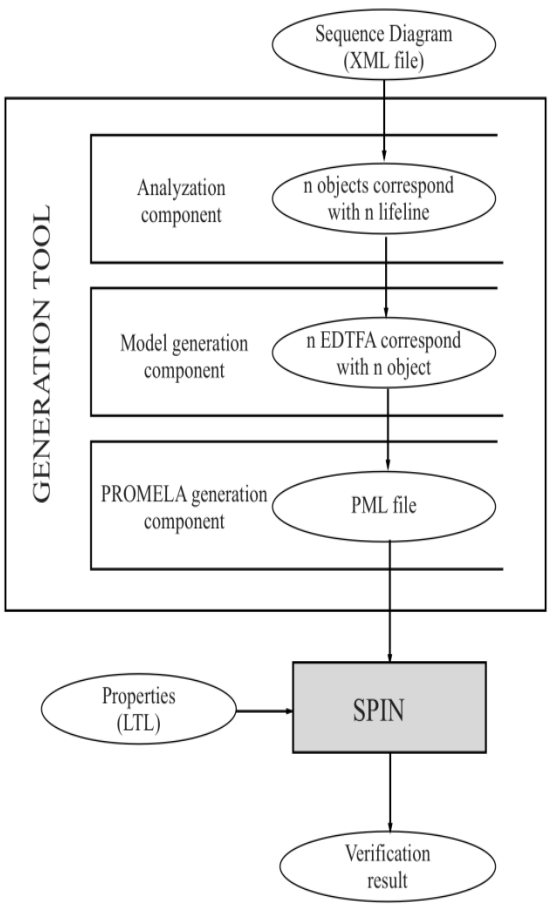

Figure 4. Architecture of PROMELA generation tool and verification process of sequence diagrams.

This tool is applied with not only simple sequence diagrams (which do not have or have only one fragment) but also with the more complex ones (which have more than one fragments or have nested fragments), then compares the results with results of the method proposed in [18]. Table 2 presents this comparison. The method proposed in this paper is more complete than the original one. This method can handle consider and ignore fragments, and specially sequence diagrams with nested fragments. The PML file received can be verified using SPIN.

To expertise the correctness and effectiveness of this tool, we tested with some sequence diagrams and individual properties. There are three systems used in testing, ATM [12], gas pumping [5], and ticket ordering. For each system, we used some properties for verification. The results are presented in Table 3. For every unsatisfied properties, SPIN can provide a counter-example.

This tool represents a completed process for modeling and verifying UML 2.0 sequence diagrams using SPIN/PROMELA. Because it does not require the formal specification of system design, this process becomes more practical for software development in software companies. However, due to the limitation of this paper, the systems used in the experiment are still simple. Therefore, the experimental results haven't been able to accurately reflex the complex of a real system design.

TABLE 2. COMPARISON BETWEEN PROPOSED METHOD AND METHOD IN [18]

No.	Fragments	[18]	Proposed Method
1	No fragment	Yes	Yes
2	Alternative	Yes	Yes
3	Loop	Yes	Yes
4	Option	Yes	Yes
5	Break	Yes	Yes
6	Parallel	Yes	Yes
7	Critical	Yes	Yes
8	Strict	Yes	Yes
9	Consider	No	Yes
10	Ignore	No	Yes
11	Sequencing	No	No
12	Negative	No	No
13	Assertion	No	No
14	Nested fragments	No	Yes

TABLE 3. EXPERIMENTAL RESULTS OF SOME SEQUENCE DIAGRAMS

System	Properties	Result
ATM	Property 1	Pass
	Property 2	Pass
	Property 3	Counter Ex.
Gas Pumping	Property 1	Pass
	Property 2	Pass
	Property 3	Counter Ex.
Ticket Ordering	Property 1	Pass
	Property 2	Counter Ex.
	Property 3	Pass

VI. CONCLUSION

We have presented a completed process of automatically modeling and verifying UML 2.0 sequence diagrams using SPIN/PROMELA. In this process, a given sequence diagram is extracted to its objects. Then, each object is represented by an EDTFA. This paper proposes a method to translate these EDTFAs into PROMELA to use the model checker SPIN in the verification process. This process has some advantages in practice where large systems need to be verified in both functional and non-functional requirements in which security property is one of the main concerns in modern software development. With the support for most of UML 2.0 fragments and especially nested fragments, it can handle large, complex sequence diagrams. Because the method is simple, and mostly automatic, it can be used in software development in IT companies. Empirically, we presented a tool to support the proposed method. This tool accepts an XML file that describes a sequence diagram as the input, analyzes and extracts objects in sequence diagram to EDTFAs. Finally, it translates EDTFAs into PROMELA specification. Then, the PROMELA specification is used in SPIN to provide verification results for given properties.

There are some limitations remaining in the paper. The correctness of the proposed method is not theoretically proven. The systems used in the experiment are simple which do not accurately reflex the complexity of a real system design in practice. Furthermore, this process is not completely automatic, still requires an additional step for reviewing and modifying PROMELA file.

In the future, we have plan to focus on proving this method in theory. In addition, we are working on a method to improve the quality of sequence diagrams presentation in PROMELA introduced in [17] in order to make this method more automatic. Furthermore, we will combine this method with other component-based verification methods to deal with state-space explosion problem. For improving the experimental utilities, we will build a GUI to support the proposed tool and test the method with more complex systems.

REFERENCES

[1]. ALAWNEH, L., DEBBABI, M., HASSAINE, F., JARRAYA, Y., & SOEANU, A., "A unified approach for verification and validation of systems and software engineering models", In 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06), pp. 409-418, 2006.

[2]. BAIER, C., KATOEN, J. P., LARSEN, K. G., "Principles of model checking", MIT Press, 2008.

[3]. CLARKE, E. M., GRUMBERG, O., PELED, D., "Model checking". MIT press, 1999.

[4]. COBLEIGH, J. M., GIANNAKOPOULOU, D., PĂSĂREANU, C. S., "Learning assumptions for compositional verification", In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, pp. 331-346, 2003.

[5]. GRØNMO, R., MØLLER-PEDERSEN, B., "From UML 2 Sequence Diagrams to State Machines by Graph Transformation". Journal of Object Technology, 10(8), 1-22, 2011.

[6]. GUELFI, N., MAMMAR, A., "A formal semantics of timed activity diagrams and its PROMELA translation", In 12th Asia-Pacific Software Engineering Conference, 2005.

[7]. H. M. DUONG, L. K. TRINH, P. N. HUNG, "An Assume-Guarantee Model Checker for Component-Based Systems", In IEEE-RIVF, pp. 22-26, 2013.

[8]. JUSSILA, T., DUBROVIN, J., JUNTTILA, T., LATVALA, T., PORRES, I., & LINZ, J. K. U., "Model checking dynamic and hierarchical UML state machines", Proc. MoDeV2a: Model Development, Validation and Verification, pp. 94-110, 2006.

[9]. KNAPP, A., WUTTKE, J., "Model checking of UML 2.0 interactions", In International Conference on Model Driven Engineering Languages and Systems, pp. 42-51, 2006.

[10]. L. C. LUAN, P. N. HUNG, "A method for model generation from UML 2.0 sequence diagrams", Proc. FAIR'9, Can Tho, pp. 619-625, 2016.

[11]. LATELLA, D., MAJZIK, I., & MASSINK, M., "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker", Formal aspects of computing, 11(6), pp. 637-664, 1999.

[12]. 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 Com. Science, 254, pp.143-160, 2009.

[13]. MIKK, E., LAKHNECH, Y., SIEGEL, M., & HOLZMANN, G. J. (1998). "Implementing statecharts in PROMELA/SPIN", In Industrial Strength Formal Specification Techniques, Proc. 2nd IEEE Workshop, pp. 90-101, 1998.

[14]. P. N. HUNG, T. KATAYAMA, "Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software", In the 15th Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, pp. 479-486, 2008.

[15]. SCHÄFER, T., KNAPP, A., MERZ, S., "Model checking UML state machines and collaborations", Electronic Notes in Theoretical Com. Sci., 55(3), pp.357-369, 2001.

[16]. SIVERONI, I., ZISMAN, A., SPANOUDAKIS, G., "Property specification and static verification of UML models", In Availability, Reliability and Security, pp. 96-103, 2008.

[17]. VAN AMSTEL, M. F., LANGE, C. F., & CHAUDRON, M. R., "Four automated approaches to analyze the quality of UML sequence diagrams". In Computer Software and Applications Conf., pp. 415-424, 2007.

[18]. ZHANG, C. & DUAN, Z., "Specification and Verification of UML 2.0 Sequence Diagrams Using Event Deterministic Finite Automata", in 'SSIRI (Companion)', IEEE Computer Society, pp. 41-46, 2011.

[19]. OMG Unified Modeling Language. [Online]. http://www.omg.org/spec/UML/2.5/PDF.

[20]. HOLZMANN, GERARD J (1997). "The model checker SPIN." IEEE Transactions on software engineering 23.5: 279.

[21]. Basic SPIN Manual. [Online]. Available: http://spinroot.com/spin/Man/Manual.html

ABOUT THE AUTHOR

PhD. Chi Luan Le

Workplace: University of Transport Technology.

Email: luanlc@utt.edu.vn

The education process: received his B.S. degree from Hanoi National University of Education (2002), M.S. and PhD. degree from University of Engineering and Technology, Vietnam National University, Hanoi (2009, 2018).

Research today: assume-guarantee verification, model-based testing, and software evolution.