Removing Ambiguous Message Exchanges in Designing Sequence Diagrams for Developing Asynchronous Communication Program

Satoshi Harauchi[*], Kozo Okano[**], and Shinpei Ogata[**]

[*]Advanced Technology R&D Center, Mitsubishi Electric Corporation, Japan
[**]Electrical and Computer Engineering, Shinshu University, Japan
[*]Harauchi.Satoshi@bc.MitsubishiElectric.co.jp
[**]{okano, ogata}@cs.shinshu-u.ac.jp

Abstract – Eliminating the reworking of designs is critical for developing software systems. Faults and errors in designs must be extracted so that they do not impair subsequent implementation or test phases. When developing communication programs, faults might linger in the programs. Simply detecting them by reviewing them is difficult, especially when designing complicated and asynchronous communication programs. In this paper, we propose a method that detects faults when designing communication programs by focusing on sequence diagrams that represent message exchanges between lifelines to remove the ambiguity about the order of the exchanges. Our method consists of the following procedures. It generates model descriptions and test expressions from sequence diagrams and executes model checking with them. Then it identifies the location of the information in the diagrams at which errors occur in model checking unless the model descriptions satisfy test expressions. Such notifications enable designers to eliminate inconsistency from their diagrams. This paper describes problems of developing sequence diagrams, our method that solves it, and its implementation with UML 2.0 as well as its evaluation. The evaluation result shows that our method is effective, even though its generation time depends on the complexity of the diagrams.

Keywords: Communication programs, Sequence diagrams, Model checking, Promela, Linear Temporal Logic, UML

1 INTRODUCTION

Removing faults and errors from software systems is critical to develop software with high reliability. Faults can be found not only in software implementation but also in its design. Detecting and removing them is much more important in the design stage than in the implementation stage because removing errors from the design stage generally takes a greater cost and effort because a design must be reworked and modified. Hence, such errors must be extracted so that they do not remain in such subsequent phases as implementations or tests.

Detecting faults is also important for developing communication programs. As communication programs become bigger, their design becomes more complicated. Since detecting them by reviewing their complicated designs is difficult, such designs must be supported to detect faults.

In this paper, we propose a method to detect faults when designing communication programs. Our method focuses on the sequence diagrams that represent asynchronous exchanges of messages between lifelines. The diagrams are complicated when complex communications are being designed. Complicated exchanges of messages frequently cause faults because ambiguity about the order of the exchanges remains in the diagrams. This ambiguity shows that the order for receiving messages could not be determined when several messages are asynchronously transmitted to a specific lifeline. Our method seeks to detect the ambiguity and remove the faults in the diagrams.

Our method consists of the following procedures. First, it generates formal descriptions written in Promela [1] from sequence diagrams. Such components as lifelines and messages described in the diagrams correspond to the Promela elements. Combined fragments, which represent such control structures as alt and loop, are also translated into Promela. Next, with Linear Temporal Logic (LTL), it generates test expressions that are obtained from every message for each lifeline. The generated expressions are used for exhaustively checking the order. Then the method executes model checking with formal descriptions and test expressions. Failing to satisfy the expressions for the formal descriptions suggests the existence of ambiguity related to the order of the messages. Our method finally identifies the position in the diagrams that cause an error in the model checking. Such identification helps designers correct the diagrams and remove the errors. Our method is reapplied from the top of the procedure after error removal, and the designers repeatedly apply it until no more errors occur.

We implement the method as a tool with UML 2.0 and evaluate two aspects. The first aspect focuses on the number of identifications generated by the tool and the time spent on the procedures for various sequence diagrams. The second goes to the diagrams applied to a specific product. The evaluation result shows that the method provides ten candidates for modifying the diagrams and four out of ten candidates are required to correct it, based on interviews with the engineers who worked on the above product.

The remainder of our paper provides detailed analysis of our method. We describe the problems for developing sequence diagrams in Section 2, and our method overcomes them in Section 3. We then describe our method's implementation and evaluation in Section 4.

2 PROBLEMS OF DEVELOPING SEQUENCE DIAGRAMS

Figure 1 shows an example of sequence diagrams. Lifelines A, B, and C asynchronously communicate with each other. The figure is used when designing communication specifications. The design is considered completed after reviewing the diagrams, and then the programs are implemented based on the diagrams.

However, after implementation, a fault might occur in Fig. 1, which shows the sequence diagrams if faults occur. “msg6” is sent from lifelines C to B after “msg3”. Lifeline B receives “msg6” before “msg5”. Nevertheless, “msg6” might reach lifeline B after “msg5,” contrary to the design intention. Lifeline B emits an error due to the specification violation.

Removing faults requires time and effort. The time depends on the causes. The fault shown in Fig. 2 cannot be detected in a unit test, but it can often be detected in integration tests. Accordingly, we must rework the design, implementation, and test. The time to repair faults increases in accordance with the number of faults and the diagram complexity.

The error in Fig. 1 was caused by the ambiguity of the sequence diagrams and indicates that the situation cannot be determined in which “msg6” reaches lifeline B. This paper describes how to remove such ambiguity in designing sequence diagrams.

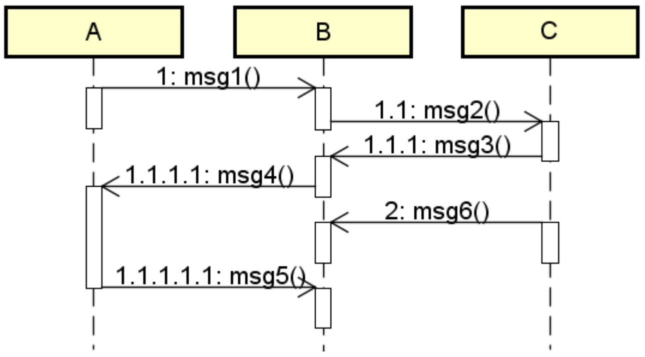


Figure 1: Example of sequence diagrams

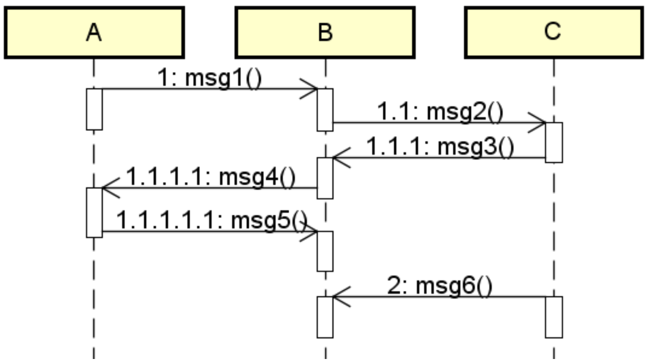


Figure 2: Sequence diagrams in case of a fault

3 PROPOSED METHOD

3.1 Outline

Our method eliminates the ambiguity of the order of messages with semi-automatic modifications of sequence diagrams. Although automatic correction is possible, our procedure provides modification candidates, enables designers to select an appropriate candidate, and corrects the diagrams with the selected candidate.

The input for the method is the diagrams written in XML. The output is the diagrams without ambiguity. The diagram specifications use UML 2.0 [1]. The diagrams allow the asynchronous representation of messages.

Our proposed method consists of these four steps shown in Fig. 3:

STEP 1: Generate formal descriptions;

STEP 2: Generate test expressions;

STEP 3: Perform model checking and generate candidates for modifying diagrams;

STEP 4: Correct the diagrams.

The contribution of this paper is STEP2 and STEP3. In STEP1 we use the existing method [2] which generates formal descriptions. STEP2 provides test expressions used for model checking, and STEP3 indicates the existence of ambiguity and candidates for removing such ambiguity.

We describe the details of each step in the following sections.

3.2 STEP 1 Generate Formal Descriptions

This step generates formal descriptions from the input. An XML is obtained with astah* professional [3]. The formal descriptions are written in Promela [4] and used by the SPIN model checker. Lima’s method [2] generates the descriptions. A lifeline and a message for each execution

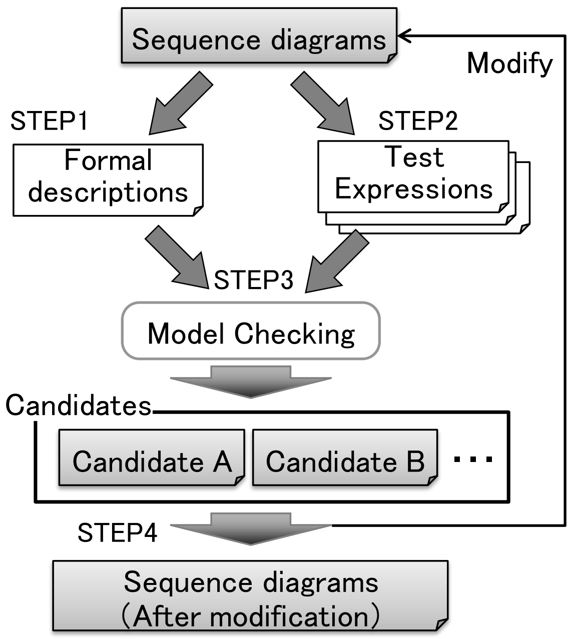
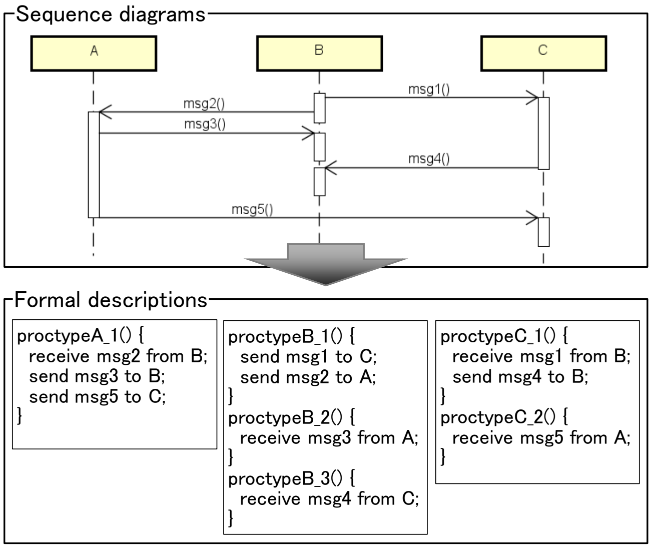


Figure 3: Overview of proposed method

Table 1: Correspondence between sequence diagrams and Promela

Sequence diagrams	Element of Promela	Description in Promela
lifeline	process	proctype {…}
message (label)	message	mtype={m1,…,mn}
message (arrow)	channel	chan chan=[1] of {mtype}; … chan chann=[1] of {mtype};
send event	send	chan!m
receive event	receive	chan?m



```
proctypeA_1() {
  receive msg2 from B;
  send msg3 to B;
  send msg5 to C;
}
```

```
proctypeB_1() {
  send msg1 to C;
  send msg2 to A;
}
proctypeB_2() {
  receive msg3 from A;
}
proctypeB_3() {
  receive msg4 from C;
}
```

```
proctypeC_1() {
  receive msg1 from B;
  send msg4 to B;
}
proctypeC_2() {
  receive msg5 from A;
}
```

Figure 4: Example of generating formal descriptions

specification respectively correspond to a process and a channel with variables defined by Promela. The description in Promela is generated based on the correspondence shown in Table 1.

Figure 4 shows an example of generating formal descriptions. Figure 5 shows a detailed example of the generated descriptions shown in Fig. 4, whose upper part shows sequence diagrams and whose lower part shows a summary of the formal descriptions generated from the diagrams. Process B_1, Process B_2, and Process B_3 are generated since lifeline B has three execution specifications. Each message is translated into two descriptions. For example, “msg1” generates one description by which B sends “msg1” to C and another one by which C receives “msg1” from B.

The generation for each execution specification maintains the order of the messages within the execution specifications. Furthermore, the generated descriptions represent the ambiguity of the order of the asynchronous messages. Lima’s method does not generate descriptions for each execution specification.

Combined fragments, which represent control structures such as alt and loop, are converted into Promela as well. We select four fragments, “alt”, “par”, “loop,” and “break,” because they are used frequently. The conversion into Promela is shown in Table 2. Description for combined fragment “par” corresponds to the diagrams shown in the right part of Fig. 6.

```
/* Auto Generated Promela File */
/* Message Declaration */
mtype = {msg1, msg2, msg3, msg4, msg5};
/* Channel Declaration */
chan to_A = [20] of {mtype};
chan to_B = [20] of {mtype};
chan to_C = [20] of {mtype};
/* Variable for send and receive */
bool send = false;
bool receive = false;
mtype msg;
/* Process Declaration */
active proctype A_1(){
   d_step{ to_A?msg2; send=false; receive=true;
   msg=msg2;}
   d_step{ send=true; receive=false; msg=msg3;
   to_B!msg3;}
   d_step{ send=true; receive=false; msg=msg5;
   to_C!msg5;} }
active proctype B_1(){
   d_step{ send=true; receive=false; msg=msg1;
   to_C!msg1;}
   d_step{ send=true; receive=false; msg=msg2;
   to_A!msg2;} }
active proctype B_2(){
   …
```

Figure 5: Detailed example of formal descriptions

Table 2: Description in Promela for control structures

Control Structures	Description in Promela
alt, break	if :: (condition 1) -> instruction 1 :: (condition 2) -> instruction 2 … :: (condition n) -> instruction n fi
loop	do :: (condition 1) -> instruction 1 :: (condition 2) -> instruction 2 … :: (condition n) -> instruction n od

par	proctype A() {
	run sub_A()
	AB_msg4?msg4; BSubB?token;}
	proctype B() {
	run sub_B()
	AB_msg4?msg4; BSubB?token;}
	proctype sub_A() {
	atomic{ AB_msg3!msg3; ASubA!token;};}
	proctype sub_B(){
	atomic{ AB_msg3!msg3; ASubA!token;};}

3.3 STEP 2 Generate Test Expressions

This step generates test expressions from the input, written in Linear Temporal Logic (LTL) expressions, which enable the representation of the system states by the changes of time. Time operators are available in addition to the conventional logical operators shown in Table 3. The expressions are used to check whether the diagrams have ambiguity about the order of the messages. Each expression is generated from two messages that are connected.

Figure 4 shows an example. Lifeline B in the sequence diagrams has four message exchanges. The item to be checked is extracted from two adjacent messages, such as "msg2" and "msg1." Since the items in the lifeline are obtained by all of the adjacent messages in relation to lifeline B, they are described as follows:

(a) Whether "msg2" was sent before "msg1" was sent;
(b) Whether "msg3" was received before "msg2" was sent;
(c) Whether "msg4" was received before "msg3" was received;

The method generates the following expressions below from (a) to (c):
(a') (send "msg2") before (send "msg1");
(b') (receive "msg3") before (send "msg2");
(c') (receive "msg4") before (receive "msg3").

The method then translates the above three items into the following test expressions:
(a'') ¬ (send "msg2") ∪ (send "msg1");
(b'') ¬ (receive "msg3") ∪ (send "msg2");
(c'') ¬ (receive "msg4") ∪ (receive "msg3").

The method generates test expressions for lifeline A and C in the same way. Two test expression are generated since they have three message exchanges. Consequently, the method generates seven test expressions in all from diagrams shown in Fig. 4.

Table 3: Time operators in LTL

Character	Symbol	Description
Xφ	◦φ	φ will be true in the next state.
Gφ	□φ	φ will always be true after this step.
Fφ	◇φ	φ will be true sometime after this step.
ψUφ	ψ ∪ φ	φ will be true sometime after this step and ψ will be true until that time.

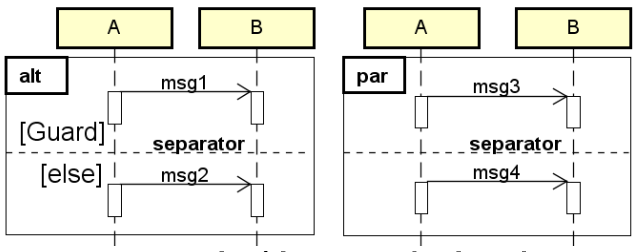


Figure 6: Example of diagrams with "alt" and "par"

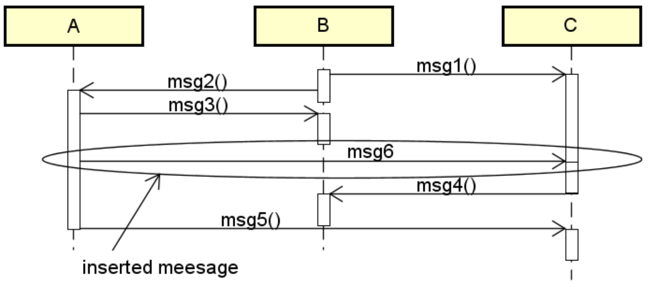


Figure 7: Example of modification

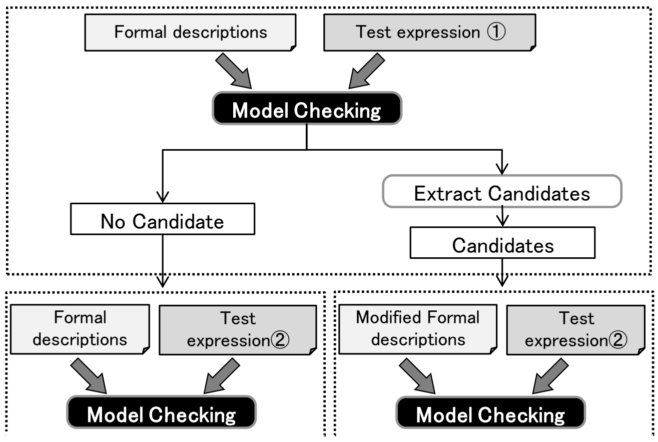


Figure 8: Procedure of generating candidates

Combined fragments are translated into expressions as well. The translation is slightly different from above. For example, "alt" or "par" in Fig. 6 has separators that are divided into operands. "msg1" and "msg2" do not need to be generated since both are executed exclusively. On the other hand, generating expressions for "par" is difficult because "msg3" and "msg4" are executed in parallel. Hence, "alt" or "par" combined fragments are ignored. Only "loop" combined fragment is dealt with for generating expressions.

3.4 STEP 3 Perform Model Checking and Generate Candidates for Modifying Diagrams

This step executes model checking with formal descriptions and test expressions. Then our method provides candidates that indicate how to modify the diagrams. Failing to satisfy the expressions suggests the presence of ambiguity. The failure result gives a pair of two messages described in the test expressions. The candidate shows diagrams with a message inserted between the pair of two messages.

Figure 7 shows an example of the modification candidate. The diagrams shown in Fig. 4 turn out to be ambiguous for the following item.

(c) Whether "msg4" was received before "msg3" was received

Therefore, lifeline A is added to the transmission of "msg6" after sending "msg3," and lifeline C is added to the reception of "msg6" before sending "msg4."

Figure 8 shows the procedure that generates candidates for all of the test expressions. First, our method selects a test expression among all of the expressions and executes model checking with formal descriptions and the selected by the

SPIN model checker. The execution moves to the following processes depending on the checking result.

If no ambiguity exists The method executes model checking with the same formal descriptions and another test expression.

If ambiguity exists The method generates a modification candidate from the test expression. The candidate is presented to a designer, and diagrams are corrected if he decides to apply it. The method then executes model checking with the corrected formal descriptions and another test expression.

The procedure is repeatedly applied and only terminates when model checking is executed for all of the test expressions.

3.5 STEP 4 Correcting the Diagrams

This step corrects the XML with the selected candidate from Section 3.4. The candidate has the information for additional messages, such as the name and the insertion place. The method corrects the definition of the messages and the information related to the lifelines. This step's procedure is completed if all the candidates indicated by the designers are reflected in the diagrams. If all the test expressions pass the model checking after the diagrams are corrected, no existence of ambiguity about the order of messages is proven for the specified diagrams.

4 EVALUATION

We implement our proposed method as a tool with Java and shell scripts to evaluate its performance and the following two aspects:

Aspect 1: Number of modification candidates generated and the time spent on the method's execution

Aspect 2: Candidate evaluation

Aspect 1 focuses on various kinds of sequence diagrams, and Aspect 2 focuses on the diagrams applied to a product. The following are the computer specifications used for the evaluation:

OS: Windows 7 Professional
CPU: Intel Xeon E5607 2.27GHz×2
Memory: 16 GB
SPIN: Version 6.3.2

The size of the state vector for the SPIN model checker was defined as 1024 bytes.

4.1 Evaluation Method

4.1.1 Aspect 1

We collected the sequence diagrams described as examples in existing researches [5]-[9] and applications [10], [11] related to sequence diagrams and produced another sequence diagrams with additional lifelines and messages for specific diagrams of the collected examples. Figure 9 shows the sequence diagrams [9]-1 produced by [9]. The square region in black dotted lines indicates the diagrams [9]. We increased the number of lifelines and messages by extending the original diagrams and applied our tool to them. We enumerated the lifelines, the messages, and the modification candidates and measured the time spent on their execution.

The measurement was executed, assuming that the designers adopted all of the candidates identified by the tool.

4.1.2 Aspect 2

We applied our tool to the diagrams used for the product. The diagrams were rewritten with astah* professional [3]. This aspect checks the ability to detect the faults shown in Fig. 2. We confirmed that the candidates generated by the tool are appropriate to be corrected.

4.2 Evaluation Results

4.2.1 Aspect 1

We applied the tool to eleven sequence diagrams. The evaluation result is shown in Table 4. The eleven diagrams consisted of seven diagrams collected from the references and four where the number of lifelines in reference [9] is increased (described as [9]-1, 2, 3, 4). Columns 1 to 3 show the information in the diagrams and columns 4 to 7 show the result applied to the tool. Columns 1, 2, and 3 respectively indicate the source of the diagrams, the number of lifelines in them, and the number of messages. Column 4 describes the number of candidates generated by the tool. Columns 5 to 7 show the execution time applied to the tool and measured for each step. Column 5 indicates the sum of the time spent on STEPs 1 and 2 since both are executed in parallel with identical input.

No significant differences can be seen in the time spent on STEPs 1 and 2. However, the time of [9]-1, 2, 3 and 4 is large. A large number of lifelines and messages increases the total amount of time. The time spent on STEP 3 becomes large as the number of lifelines or messages in the diagrams increases. The more candidates, the more time will be spent on STEP 4, although no major differences can be observed in the time.

The time spent on STEP 3 in [9]-4 is much smaller than in [9]-3, although the number of lifelines and messages are very large, because the model checking in that case could not be executed due to insufficient memory. Hence, the number of candidates became zero.

We also applied our tool to three more sequence diagrams with combined fragments. The evaluation result is shown in Table 5. The diagrams were extracted from the references. Columns 1 to 6 in Table 5 are the same as in Table 4. Column 7 describes the number of fragments in the diagrams.

The result shows that the time spent on STEPs 1 and 2 was different between the presence and the absence of the combined fragments. This difference reflects the time difference to generate formal descriptions, not test expressions.

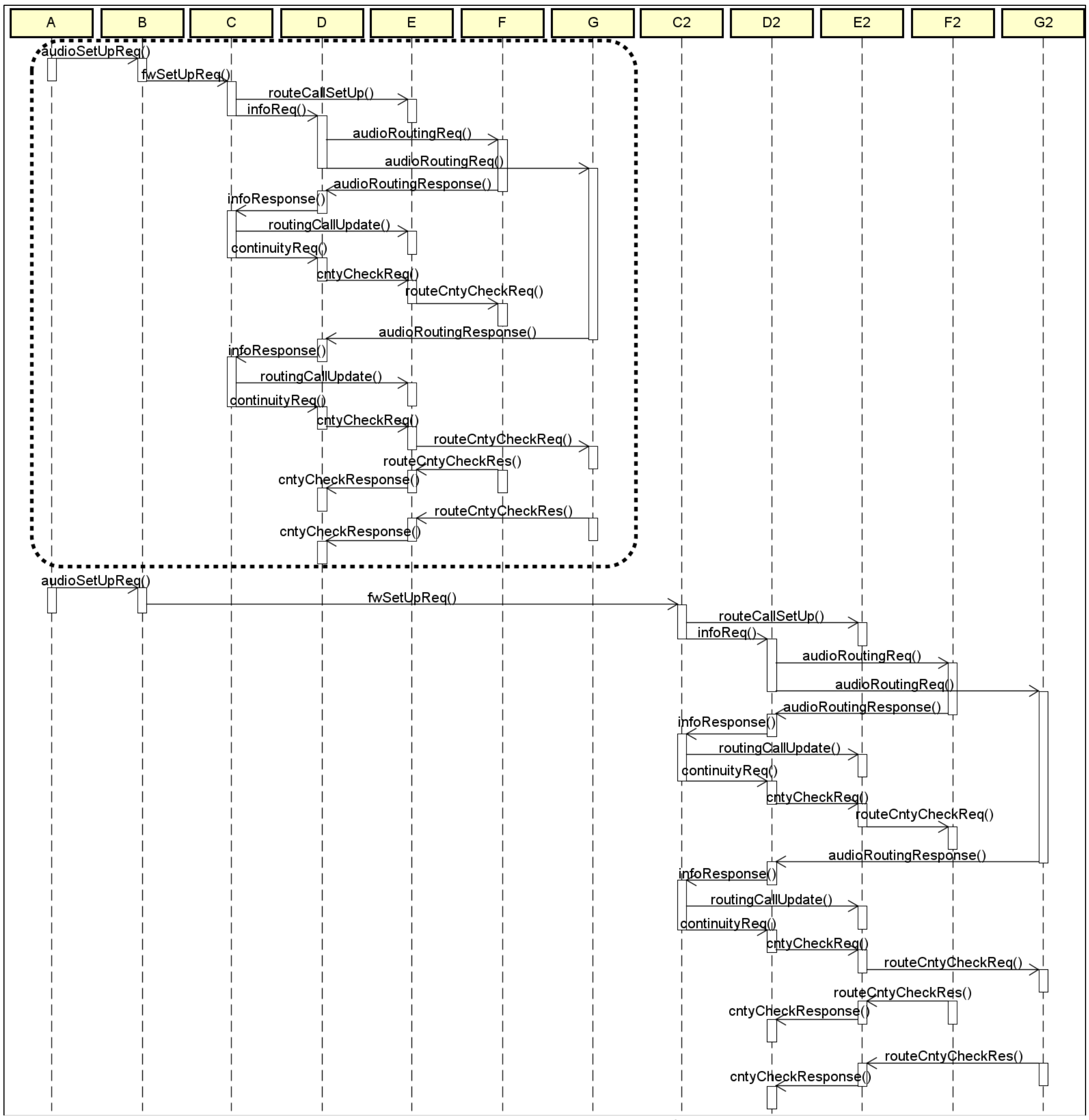


Figure 9: Sequence diagrams [9]-1 produced by [9] for evaluating Aspect 1

Table 4: Evaluation results for Aspect 1

	Lifelines	Messages	Modification candidates	Time spent on execution(seconds)		
				STEP 1,2	STEP 3	STEP 4
[5]	7	11	3	0.44	20.21	0.43
[6]	4	12	2	0.42	25.19	0.41
[7]	3	8	0	0.39	15.55	0.41
[8]	3	4	0	0.37	5.96	0.40
[10]	5	6	1	0.42	8.95	0.41
[11]	6	24	0	0.49	56.15	0.42
[9]	7	22	6	0.46	52.08	0.48
[9]-1	12	44	9	0.58	124.77	0.45
[9]-2	22	88	18	0.98	9136.59	0.52
[9]-3	32	132	24	0.97	14054.17	0.56
[9]-4	37	154	0	1.20	744.73	0.56

Table 5: Evaluation result for Aspect 1 with combined fragments

	Lifelines	Messages	Modification candidates	Time spent on execution(seconds)		Combined fragments
				STEP 1,2	STEP 3	
[12]	4	7	3	1.40	1.3	0
[13]	3	10	4	5.60	16.3	2 (loop, alt)
[6]	4	9	1	5.20	16.0	2 (loop, par)

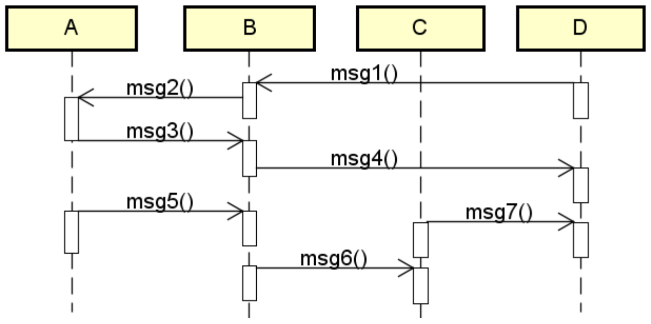


Figure 10: Example of diagrams used for Aspect 2

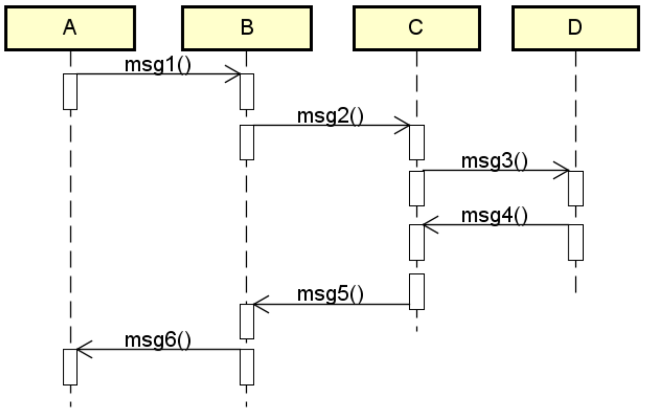


Figure 11: Example of diagrams used for Aspect 2

Table 6: Evaluation result for Aspect 2

	Lifelines	Messages	Modification candidates	Time spent on execution (seconds)		
				STEP 1,2	STEP 3	STEP 4
Figure 10	4	7	5	0.40	14.04	0.62
Figure 11	4	6	5	0.41	12.15	0.52

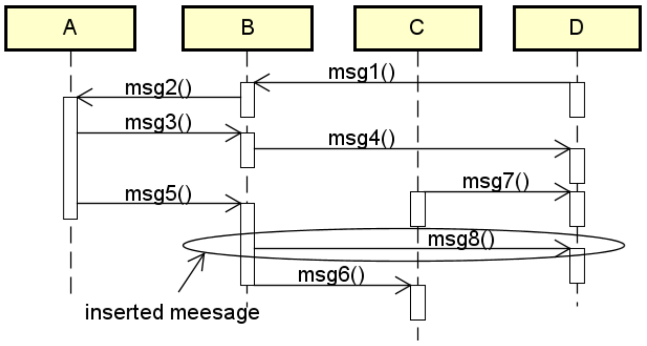


Figure 12: A modification candidate 1

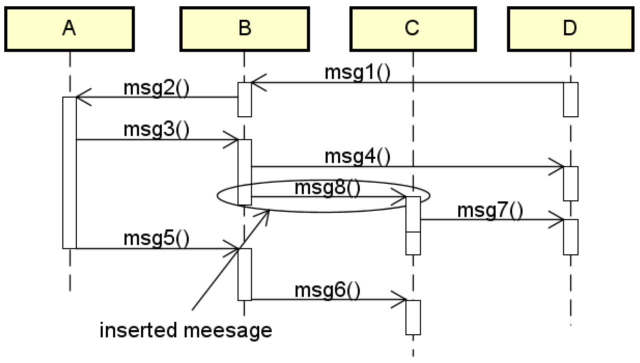


Figure 13: A modification candidate 2

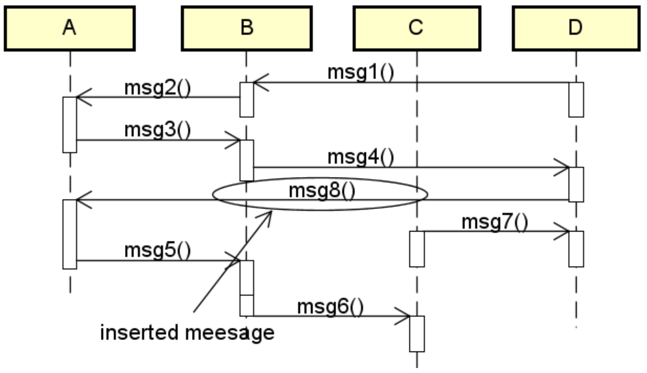


Figure 14: A modification candidate 3

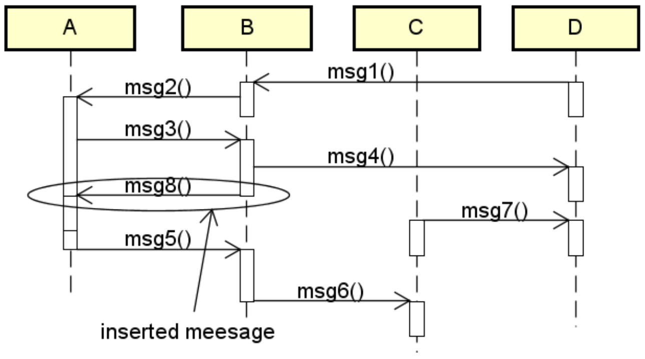


Figure 15: A modification candidate 4

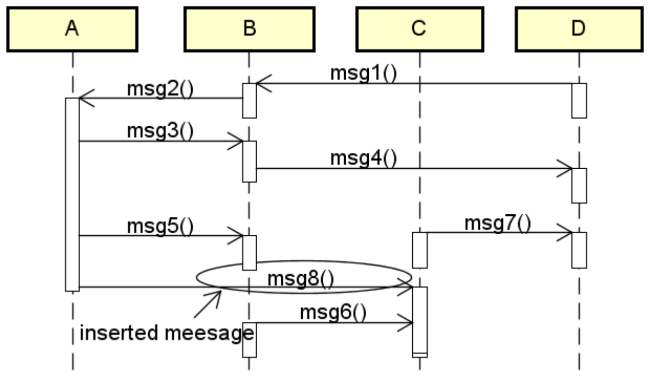


Figure 16: A modification candidate 5

Table 7: Removed ambiguity for each candidate

Candidate	Ambiguity
Figure 12	“msg7” might reach lifeline D after “msg6”
Figure 13	“msg7” might reach lifeline D before “msg4”
Figure 14	“msg5” might reach lifeline B before “msg4”
Figure 15	“msg5” might reach lifeline B before “msg3”
Figure 16	“msg6” might reach lifeline C before “msg7”

4.2.2 Aspect 2

Some of the diagrams used for the product were supplied by factories for evaluating Aspect 2. We selected two diagrams from the supplied diagrams and applied the tool to them where ambiguity about the order of the messages might exist. The applied diagrams are shown in Fig. 10 and Fig. 11. The application result is shown in Table 6 whose columns are the same as in Table 4.

Next we describe all of the candidates obtained by our tool for Fig. 10. The candidates are shown from Fig. 12 to Fig. 16. For instance, Fig. 12 indicates the modification that inserts “msg8”. The candidate is generated from the ambiguity that “msg7” might reach lifeline D after “msg6”.

Table 7 shows the removed ambiguity for all of the candidates. The result shows that our proposed method can detect the ambiguity and generate the candidates for each ambiguity for the sequence diagrams used for the products.

We requested the engineers who developed the product to check the ten candidates including from Fig. 12 to Fig. 16. We confirm their validity by selecting one appropriate answer from the following three options:

(1) This modification must be done.
(2) This modification does not need to be done.
(3) This modification should not be done.

Option (1) was selected for four candidates including Fig. 12. (2) was selected for five candidates and (3) for one candidate.

The answers show that the method detect necessary candidates, but several candidates are of no use. Unnecessary candidates are caused by constraints for modifying sequence diagrams. Figure 10 and Fig. 11 has the lifeline which corresponds to the equipment procured externally. It is impossible to modify sequence diagrams related to such equipment. Hence, the engineers selected answer (2) or (3). We should consider the constraint in order to exclude unnecessary candidates when applying to the design for products.

4.3 Evaluation Validity

In Aspect 1 we collected the sequence diagrams from existing researches and tools and evaluated the number of modification candidates and the time spent on the execution by applying the tool to the diagrams. In this paper we only applied the tool to eleven sequence diagrams. Since the number of lifelines and messages written in the diagrams is limited, we might obtain a different result when applying the tool to large-scaled diagrams.

In Aspect 2 we confirmed the possibility of detecting faults and generated modification candidates from diagrams with ambiguity developed for the product. We only used two diagrams for our evaluations. We must obtain a large variety of diagrams for various products and evaluate them to acquire more general results.

5 RELATED WORKS

Lima et al. proposed a method that generates Promela from sequence diagrams and detects faults with model checking [2]. This method shows representation written in Promela for almost all of the diagrams described in UML 2.0. They implement this method as an eclipse plugin and confirm fault detection by giving appropriate test expressions.

Miyamoto et al. proposed a method that converts the specifications of software written in state diagrams and deployment diagrams in Promela representation [14]. The input for both diagrams is XML produced by astah* professional. Their method generates Promela by translating instances in the deployment diagrams into processes and translating the transitions in state diagrams into processing that executes each process. Converting the patterns of the above specifications in UML into LTL expressions enables the execution of SPIN model checking without describing complicated expressions.

Nagata et al. proposed a method that generates communication programs from the specifications of communication protocols described with sequence diagrams [15]. Their method, which defines the protocols with the diagrams and a format that represents the content of the messages, generates programs from the above definitions. The generation derives exception handling from fault tree diagrams and appends it to the programs in normal processing. The method reduces the overlooking of exception handling required for the occurrence of exceptions for communication programs.

Tiwari et al. proposed a method that generates test cases with activity diagrams that describe software specifications [16]. Their method obtains the conditions under which a system terminates normally from diagrams that represent the processing flow. Their method acquires fault tree diagrams by reversing the conditions and generates test cases where the system terminates both normally and abnormally.

Kaleeswaran et al. proposed a method that detects faults from programs and test suites and shows candidates for correcting the faults [17]. Their method modifies programs based on points specified by toolset Zoltar [18] which automatically localizes faults. Since the modification is then executed by selecting the candidates, it enables semi-automatic corrections.

6 CONCLUSION

This paper proposed a method that detects faults when designing sequence diagrams that describe the asynchronous exchanges of messages. Our method transforms formal descriptions written in Promela and test expressions written in Linear Temporal Logic (LTL) from sequence diagrams and executes model checking for all of the expressions with the descriptions. When an error occurs in an execution, it provides information in diagrams, enabling designers to remove the faults and protect consistency.

We implemented and evaluated our method with two aspects. In the first aspect, we measured the amount of information and the time spent on our method's execution. In the second one, we applied our method to the diagrams used by a product. The application generated ten pieces of information and evaluated their validity. According to interviews with engineers, about 40% of the information is effective for correcting the diagrams. Future work will apply our method to various developments of diagrams and increase the number and the kinds of candidates.

REFERENCES

[1] "UML2.0," http://www.omg.org/spec/UML/2.0/, retrieved on September 13, 2016.

[2] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi, "Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages," Electronic Notes in Theoretical Computer Science, vol. 254, pp. 143-160 (2009).

[3] "astah* professional," http://astah.net/editions/ professional, retrieved on September 13, 2016.

[4] G. J. Holzmann, "The model checker SPIN," IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279-295 (1997).

[5] P. Baker, P. Bristow, C. Jervis, D. King, R. Thomson, B. Mitchell, and S. Burton, "Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams," Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 50-59 (2005).

[6] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analyzable Petri Net models," Proceedings of the 3rd International Workshop on Software and Performance, pp. 35-45 (2002).

[7] D. Harel and S. Maoz, "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams," Software & Systems Modeling, vol. 7, no. 2, pp. 237-252 (2008).

[8] H. Shen, R. Krishnan, R. Slavin, and J. Niu, "Sequence Diagram Aided Privacy Policy Specification," IEEE Transactions on Dependable and Secure Computing, pp. 381-393 (2014).

[9] B. Mitchell, "Characterizing Communication Channel Deadlocks in Sequence Diagrams," IEEE Transactions on Software Engineering, vol. 34, no. 3, pp. 305-320 (2008).

[10] "Lucidchart," https://www.lucidchart.com/, retrieved on September 13, 2016.

[11] "tracemodeler," http://www.tracemodeler.com/, retrieved on September 13, 2016.

[12] H. Shen, R. Krishnan, R. Slavin, and J. Niu, "Sequence Diagram Aided Privacy Policy Specification," IEEE Transactions on Dependable and Secure Computing, no. 99 (2014).

[13] D. Harel and S. Maoz, "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams," Software & Systems Modeling, vol. 7, no. 2, pp. 237-252 (2008).

[14] N. Miyamoto and K. Wasaki, "Automatic Conversion from the Specification on UML Description to PROMELA Model for SPIN Model Checker," Forum on Information Technology, vol. 9, no. 1, pp. 311-314 (2010).

[15] T. Nagata, S. Harauchi, M. Kitamura, T. Yamaji, and Y. Ueno, "A Method to Create Network Communication Programs by Deriving Exception Handling from Fault Tree Diagram," Forum on Information Technology, vol. 11, pp. 45-48 (2012).

[16] S. Tiwari and A. Gupta, "An Approach to Generate Safety Validation Test Cases from UML Activity Diagram," Proceedings of the 20th Asia-Pacific Software Engineering Conference, pp. 189-198 (2013).

[17] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated Synthesis of Repair Hints," Proceedings of the 36th International Conference on Software Engineering, pp. 266-276 (2014).

[18] T. Janssen, R. Abreu, and A. J. C. van Gemund, "Zoltar: A Toolset for Automatic Fault Localization," Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 662-664 (2009).

(Received October 7, 2016)
(Reviced May 3, 2017)

Satoshi Harauchi received BE and ME degrees in Information Sciences from Kyoto University in 1996 and 1998, respectively. Since 1998, he has been at the Advanced technology R&D center of Mitsubishi Electric Corporation and is currently interested in software engineering for social infrastructure system. He is a member of IEICE and JSASS.

Kozo Okano received BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an associate professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2015, he has been an associate professor at the Department of Computer Science and Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, and IPSJ.

Shinpei Ogata is an assistant professor of the Graduate School of Science and Technology in Shinshu University, Japan. He received a PhD from Shibaura Institute of Technology, Japan in 2012. His current research interests include model-driven engineering for information system development. He is a member of IEEE, ACM, IEICE, and IPSJ.