A Method for Automated Test Cases Generation from UML Models with String Constraints

Thi Dao Vu[1], Pham Ngoc Hung[2], and Viet Ha Nguyen[2]

[1] Academy of Cryptography Techniques
141 Chien Thang str., Thanh Tri dist., Hanoi, Vietnam
vtdao@bcy.gov.vn or vuthidao@gmail.com
[2] VNU-University of Engineering and Technology
hungpn@vnu.edu.vn and havn@vnu.edu.vn

Abstract. This paper proposes an automated test cases generation method from sequence diagrams and class diagrams with string constraints. The method supports UML 2.0 sequence diagrams including twelve combined fragments. An algorithm for generating test scenarios are developed to avoid test paths explosion without having data sharing points of threads in parallel fragments or weak sequencing fragments. Test data are also generated with solving constraints of string variables. We standardize string constraints and equations at the boundary of variables that are input formula of Z3-str solver. Comparing with the current approach of the solver, some preprocessing rules are extended for other operations such as charAt, lastindexOf, trim, startsWith and endsWith. If a result of the Z3-str is SAT, test data of each test scenario are generated to satisfy the constraints with boundary coverage. A tool is implemented to support the proposed method, and some experiments are also presented to illustrate the effectiveness of the tool.

1 INTRODUCTION

Model-based testing plays a significant role in research and practice due to great benefits. There are some approaches for model-based testing: test data generation, test cases generation from behavior models and test scripts generation from abstract tests [6]. Test data generation from solving constraints has focused on primitive data type of variables. However, there are many applications being faulty in doing strings processing. In addition, one of major approaches is generation of test cases from Unified Modeling Language (UML) models. In this approach, an intermediate model helps to generate the control flow sequences. There are three options to choose the intermediate models [3] such as activity diagram, control–flow graph (CFG) [8, 5] and Colored Petri Nets. The test scenarios which are abstract test cases help to find errors during implementation of software systems.

Many works have been proposed in order to show the approach. However, an approach [1] did not address different types of combined fragments, especially in case of nested combined fragments. And the method in [1] did not also generate test data. A method [5] dealt with five interaction fragments such as loop, alt, opt, break and parallel fragments in UML 2.0 sequence diagrams. Moreover, in [8] a method was developed for eight kinds of combined fragments describing control flow of systems, and the method only solved test data generation with numeric data type.

There are many string solvers such as REX [7], DPRLE [2] and HAMPI [4] that only use string operations [9], but many non-string operations in applications are also popular. Moreover, the string operations interact with the non–string operations that cause errors. An analyzing of string-only will be the shortage of pure integer constraints. There are many solvers converting string-to-integer constraints that are not precise enough. Therefore, a Z3–str solver [9] is used by supporting of a combined logic both strings and non-string operations.

The paper proposes a method in order to generate automatically test cases from sequence diagrams and class diagrams with string constraints. This method is to solve all twelve kinds of fragments in UML 2.0. An algorithm for generating test scenarios is developed to avoid test paths explosion without having data sharing points of threads in parallel (par) or weak sequencing fragments (seq). String constraints of each test scenario and equations at the boundary of variable are converted into input of Z3–str solver. If output of the solver is SAT, a possible model is given. Test data are given with satisfying the constraints and boundary coverage from the possible model. Comparing with Z3–str solver, some preprocessing rules are extended for other operations such as charAt, lastindexOf, trim, startsWith and endsWith. A tool is implemented to support the proposed method. Some experiments are illustrated the effectiveness of the tool.

The paper is organized as follows: Section 2 mentions transforming UML sequence diagrams and class diagrams into CFG, Sect. 3 describes the algorithm of test scenarios generation, Sect. 4 presents solving string constraints. A tool to implement the proposed method and some experiments to validate its feasibility and effectiveness are shown in Sect. 5. We conclude the paper and discuss future works in Sect. 6.

2 CONTROL FLOW GRAPH GENERATION

Test sequences generation from UML 2.0 models needs an intermediate model. Elements of the model will be processed easily. A CFG is chosen in our approach. The proposed technique of CFG generation requires UML diagrams in xmi file. To solve all twelve fragments in UML 2.0 sequence diagrams, CFG generation is extended [8] for the remaining four fragments: ignore, consider, negative and assertion. String constraints of variables are derived from class diagrams and conditions of sequence diagrams which are solved to generate test data.

The definitions of CFG, a block node (BN), a decision node (DN), a merge node (MN), a fork node (FN) and a join node (JN) are mentioned in [8]. We use again Algorithm 1 in [8] for generating CFG from analyzing the queue, and develop Algorithm 2 for ignore, consider, negative and assertion fragments (shown in Fig. 1 and detailed description in below Algorithm 2 in Sect. 7).

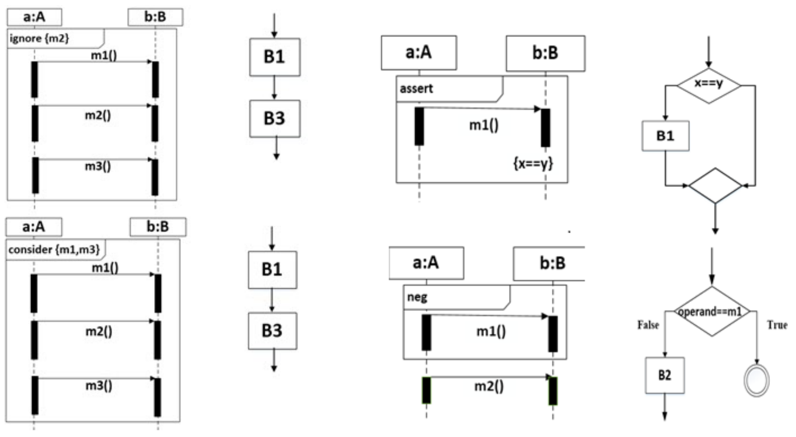

Fig. 1. General structure of CFG (for ignore, consider, assert and neg fragments).

3 TEST SCENARIOS GENERATION

Input to the test scenarios generation is a CFG. After constructing the CFG, the CFG is traversed automatically to generate the test scenarios which sat isfy coverage criteria. The test scenarios denote abstract test cases which are paths starting from the initial node to a final node. Each given test scenario begins with the initial node (*in*) of the CFG.

Algorithm 1 Generating the test scenarios

Input: Control-flow Graph G with initial node *in* and final nodes are fn_i
Output: T is a collection of test scenarios, t is a test path

```
1: $T = \emptyset$; $t = \emptyset$; $queue = \emptyset$;
2: $curNode = in$; //current node starts from $in$
3: repeat
4:     t.append(curNode);
5:     move to next node
6:      if ($curNode == DN$ and decision==TRUE) then
7:          Append true part of BN up to MN in t
8:      else
9:          Append false part of BN up to MN in t
10:     end if
11:     if ($curNode == FN$) then
12:         active all nodes of threads; nodes = ready;
13:         put a beginning node x of queue; x= waiting;
```

```
14:         repeat
15:             remove front node y of queue; y=processed;
16:             t.append(y);
17:             The neighbour(z) of y having ready status add to the end of queue;
18:             z = waiting;
19:         until($queue$ is empty)
20:     end if
21:     if ($curNode == fn_i$) then
22:         T = T + {t};
23:     end if
24: until Graph end
```

The selection coverage criterion of sequence diagrams is used to cover the diagrams during testing (the test scenarios ensure that each branch of selector modeled is traversed at least once). Depth-first search (DFS) algorithm was used in [5] to generate test scenarios, but the method did not address the issues of the synchronization and data safety. A method in [8] solved this issue to avoid test explosion by selecting switch points of threads in par or seq fragments. However, there are many applications without having sharing data points of threads in these fragments. In this case, an Algorithm 1 is developed (GenerateTestScenarios) to traverse CFG using both DFS and breadth-first search (BFS) algorithm.The BFS is useful for traversing CFG if current node is a fork node. Besides, the proposed method traverses those nodes of threads in case of parallel by BFS to avoid finding the sharing data points of threads in par or seq fragments. The remaining of CFG is traversed by using DFS. Therefore, it avoids wasting time of finding the data sharing points and test paths explosion.

4 SOLVING STRING CONSTRAINTS

The test scenarios obtained denote the sequences of messages. The sequence is a feasible sequence of messages if we find test data (test input) to satisfy all constraints along the scenario. Many current researches [8, 5] solve the equations to find values that satisfy these constraints. However, it generates test data in case of numeric data type and rarely considers string constraints. Using a Z3–str solver generates test data if constraints are satisfiable. The input of the solver are all the constraints along the scenario and equations at the boundaries of the domains of variables. If output of the solver is SAT that means all constraints are satisfiable, a possible model is given. We take examples from the possible model, and test data are satisfiable with boundary coverage.

4.1 Input formula of Z3–str solver

Z3–str can handle a boolean combination of atomic formulas, it is converted into conjunction of literals. We will use an example of string solving for Z3–str. Consider string constraints: $c1, c2, x : String$; $vi1 : Integer$; $c1 = c1.concat(''te'')$;

$c2 ='' aaaa\_efg\_bbbb\_efg''$;$x = c1.concat(c2)$;$vi1 = x.indexOf(''efg'')$; $vi1 \geq 4$;
The core treats the string operations as five independent boolean variables ($e1$,$e2$,$e3$,$e4$ and $e5$) and tries to assign values to them.
$e1 : c1 = c1.concat(''te'')$; $e2 : c2 ='' aaaa\_efg\_bbbb\_efg''$;
$e3 : x = c1.concat(c2)$; $e4 : vi1 = x.indexOf(''efg'')$; $e5 : vi1 \geq 4$;
Consider the string constraints above, the input formula of Z3–str is converted as follows assert ($e1 \bigwedge e2 \bigwedge e3 \bigwedge e4 \bigwedge e5$). Each operation above is transformed into: (assert (concat c1 ”te”)); (assert (= c2 ”aaaa_efg_bbbb_efg”)); (assert (= x (concat c1 c2)); (assert (=vi1 (indexOf x ”efg”))); (assert ($\geq vi1$ 4))
If the output of Z3–str of their respective input is satisfiable, values of variables are given by using get–model. A few good data values are chosen as test inputs when there are a number of possible input values using boundary coverage. There are a lot of faults in the system under testing that are located at the frontier between two functional behaviors. In our approach, constraints added at a boundary point of predicates are input of Z3–str (input of Z3-str is added by $vi1 == 4$). Therefore, test data are generated and satisfiable boundary coverage.

4.2 Improving preprocess rules for other operations

In [9] plug-in of Z3–str supports the string operations: string equation, concatenation, length, substring, contains, indexof, replace and split. They have three primitive operations: string equation, concatenation and string length. That method reduces other string operations to an equivalent formula based on above primitives. They performed pre–processing to translate substring, contains, indexOf, replace and split operations into formulas using concatenation and length operations. Extension of some rules of the preprocessing is presented for other operations such as charAt, lastindexOf, trim, startsWith and endsWith when comparing to [9]. The rules are converted into the primitive operations which are as follows,

Table 1. Pre–processing rules for other operations

Expression	**Rule** **New Formula**
c=x.charAt(i)	$charAt(x, i) = c \rightarrow x = x_1.t.x_2 \bigwedge t = c \bigwedge length(x_1) = i$
i= $x_1.lastIndexOf(x_2)$	$lastIndexOf(x_1, x_2) = i \rightarrow (x_1 = x_{s1}.x_{s2}.x_{s3}) \bigwedge (i = -1 \bigvee i \geq 0) \bigwedge ((i = -1) \leftrightarrow (\neg contains(x_1, x_2)) \bigwedge ((i \geq 0) \leftrightarrow (i = length(x_{s1}) \bigwedge x_{s2} = x_2 \bigwedge (\neg contains(x_{s3}, x_2)))$
$x_2 = x_1.trim$	$trim(x_1, x_2) \rightarrow (x_1 = x_{s1}.x_{s2}.x_{s3}) \bigwedge (x_{s2} = x_2) \bigwedge ((x_{s1}=" ") \bigvee (x_{s3}=" "))$
j= $x.startsWith(x_t, i)$	$startsWith(x, x_t, i, j) \rightarrow (x = x_1.x_2.x_3) \bigwedge (j = 1 \bigvee j = 0) \bigwedge ((j = 1) \bigwedge x_2 = x_t \bigwedge length(x_1) = i) \bigwedge ((j = 0) \bigwedge (\neg contains(x, x_t))$
i= $x.endsWith(x_t)$	$endsWith(x, x_t, i) \rightarrow (x = x_1.x_2.x_3) \bigwedge (i = 1 \bigvee i = 0) \bigwedge ((i = 1) \bigwedge x_2 = x_t \bigwedge \neg contains(x_3, x_t)) \bigwedge ((i = 0) \bigwedge \neg contains(x, x_t))$

charAt. takes two arguments x, i and the result is c. This operator returns the character c located at the specifying i of string x. The indexing of the string x starts from zero. A formula of charAt can be converted with concatenation and string length operations. Particularly, we break the argument x into three pieces x_1, t, x_2, and assert the middle piece t which equals to the return character c. We assert the lengths of x_1 to respect the position of constraints.

lastIndexOf. if the string x_2 argument occurs one or more times as a substring within string x_1, then it returns the index of the first character of the last substring x_2. If it does not occur as a substring, -1 is returned. We break x_1 into three pieces x_{s1}, x_{s2} and x_{s3}. The result value i options: if and only if string x_1 does not involve x_2, i is -1. Otherwise, if and only if x_{s2} equals to x_2, its predecessor x_{s3} does not contain x_2 , and i equals to the length of x_{s1}.

trim. method returns x_2 that is a copy of the string x_1 and omits leading and trailing whitespace. We break x_1 into three pieces x_{s1}, x_{s2} and x_{s3}, and assert the middle piece x_{s2} which equals to the return string (x_2). We assert x_{s1} or x_{s3} to respect whitespace.

startsWith. it tests whether the string x_t is a substring of string x and x_t starts with the specified prefix beginning (i). It returns true ($j = 1$) if the character sequence represented by the argument (x_t) is a prefix of the character sequence represented by this string x; false ($j = 0$) otherwise. The first argument x is broken into three pieces x_1, x_2 and x_3. The result value j options: if and only if string x does not contain x_t, j is 0. Otherwise (j is 1), we assert the string x_2 and length of x_1 to respect the string x_t and the specified index (i).

endsWith. this method returns true ($i = 1$) if the character sequence represented by the argument (x_t) is a suffix of the string (x), else false ($i = 0$). We break x into three pieces x_1, x_2 and x_3. The result value i options: if and only if string x does not contain x_t , i is 0. Otherwise, if and only if x_2 equals to x_t, its predecessor x_3 does not contain x_t.

5 EXPERIMENTS

This section proposes a tool, SequenceString which is developed to support the proposed method. A case study is conducted to examine the method. And then some experiments are analyzed about performance and errors found in test scenarios that is used to evaluate its effectiveness.

5.1 Tool Support

In this section we discuss the results by implementing the proposed method. The method is implemented using JAVA and JDK version 1.8. Our method is developed for generating test cases automatically from UML sequence diagrams and string constraints, data type of variables from class diagram. The architecture of the tool is shown in Fig. 2. The implemented tool is available at the site[3].

The Tool consists of 2216 lines of code and has the following functionality:

[3] http://www.uet.vnu.edu.vn/∼hungpn/SequenceString/

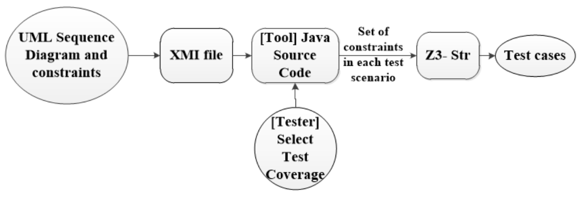
UML Sequence Diagram and constraints
XMI file
[Tool] Java Source Code
Set of constraints in each test scenario
Z3- Str
Test cases
[Tester] Select Test Coverage

Fig. 2. Figure showing architecture of SequenceString.

1. Preprocessing: Enterprise Achitect ver.11 is used to produce the UML design artefact. The tool imports the UML sequence diagrams and constraints, data type of variables from class diagram (in XMI file).
2. Generating test scenarios: for each coverage criterion, the tool generates test scenarios from CFG and gives constraints along with each test scenario.
3. Generating test data: with set of constraints along with the test scenario, it is converted into input formula of Z3–str solver. The constraints include numeric constraints, string constraints and equations of variables satisfying at boundary value. If the output of Z3–str solver is SAT, a possible model is given. Test data of each test scenario are generated from taking examples in the possible model, and it is satisfiable boundary coverage.

5.2 Case Study

In this section, the test cases generation is illustrated from UML 2.0 sequence diagram and string constraints, data type of variables from class diagram. Fig. 3 shows an example that has input string s, and data types of variables are given that are: String s, s1; int i = s.lastIndexOf(','); int f = Long.parseLong(s); BigDecimal d1 = BigDecimal(-1); s1 =s.substring(i+1);int x= Integer.parseInt(s1); There are six test scenarios in accordance with selection coverage criterion in the example. Test scenario passing $m5()$ is considered. Firstly, it checks whether input string s starts with '-' (that can represent a negative number). Then if s is of a format (the string involves beginning with '-', followed by at least one digit,and comma, lastly 3 digits). After that, $(i = s.lastIndexOf(','))! = -1$ means that it checks whether comma appears in s. With String s1 =s.substring(i+1) if string s1 which is string after comma is substring of string s, and then s1 is converted into the integer x. Finally, it continues by checking whether x is greater than or equals to 100. If the condition is satisfiable, test scenario passes $m5()$.
When sequence diagram (XMI file) and data type of variables are input into our tool, CFG is generated. When we click one test scenario, the details of test path show a set of constraints along with the test scenario. For example, this path (passing $m5()$) has conjunction of some constraints which are as follows,
s.charAt(0)=='-';s.notMaches("-\d+,\d{3}"); i = s.lastIndexOf(',');
i !=−1; s1 =s.substring(i+1); x= parseInt(s1); $x \geq 100$
Equation at boundary value is x ==100. It is converted into input formula of Z3–str. If output of Z3–str is SAT, variable s of this test scenario is "-1,000,100".

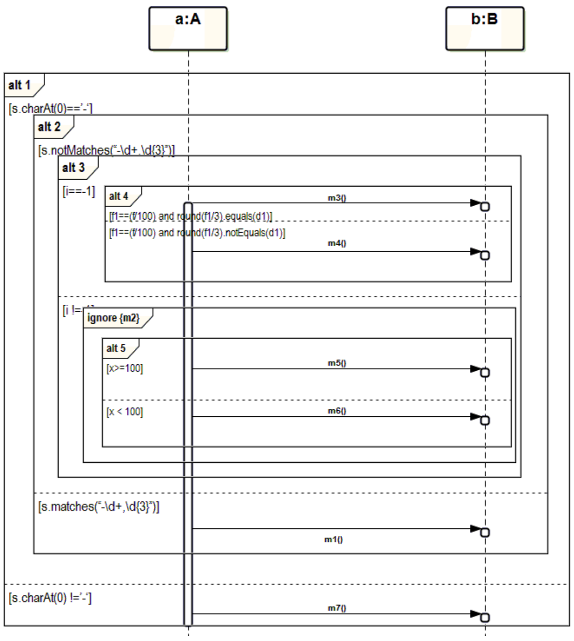
a:A
b:B
alt 1
[s.charAt(0)=='-']
alt 2
[s.notMatches("-\d+.\d{3}")]
alt 3
[i==-1]
alt 4
m3()
[f1==(f/100) and round(f1/3).equals(d1)]
[f1==(f/100) and round(f1/3).notEquals(d1)]
m4()
[i !=
ignore {m2}
alt 5
[x>=100]
m5()
[x < 100]
m6()
[s.matches("-\d+,\d{3}")]
m1()
[s.charAt(0) !='-']
m7()

Fig. 3. The sequence diagram for a checking information of web application.

5.3 Evaluation

Our evaluation consists of two experiments. In the first experiment, three applications have some functions with string constraints and relational operators. We compare the percentage of errors found in applications which are used by our proposed method and random test data generation. In the second experiment, we compare the performance of tool with test data generation in the same test scenario when to use or not to use the preprocessing rules for some operations such as charAt, lastIndexOf, trim, startsWith and endsWith. All experiments are run on an Intel Core i3- 6100U CPU 2.30 GHz with Ram 4 GB.

Table 2. Comparison errors found in applications

Application	Description	Inputs	Errors found of our method (%)	Errors found of random test data (%)
A	Checking information of user registration	3(strings)	100	42.5
B	Business ordering	5(strings)	100	36.5
C	Insurance registration	4(strings)	90	35.6

Comparison errors: in the functions of three applications, errors are injected at the points of boundary. We perform checking information of user registration function, business ordering and insurance registration. The user registration function has three string variables and two relational operators. The business ordering has five string variables and three relational operators. The last function has four string variables and zero relational operator. From sequence diagrams of the applications, our tool is used to generate test cases. We compare errors found in the same test scenarios but test data are generated by our method and random test data generation. With the proposed method, the fourth column is the percentage of errors found in test scenarios in total errors which inserted into functions. The fifth column is the percentage of errors in the same test scenarios with random test data generation. Therefore, the ability of our method in terms of finding errors is better than that of random test data.

Comparison performance: because Z3-str is open- sourced. We are looking for the communication between Z3-str and the string theory to improve performance. Some rules of the preprocessing are extended for other operations such as charAt, lastindexOf, trim, startsWith and endsWith when comparing with [9]. We use 9 test cases of the checking information of user registration. Then, we run both 30 times for each test scenarios and take average of the execution time in Z3–solver and Z3–solver with precessing rules of some operations. We can see Z3–str with precessing rules of some operations is faster than Z3–str in case of charAt, lastindexOf, trim, startsWith and endsWith operations.

Table 3. Comparison performance with applying preprocessing rules

Fragment	Inputs	Time of Z3-str (s)	Time of Z3-str with preprocessing rules (s)
concat	8	0.035	0.035
indexOf	12	0.055	0.055
charAt	12	0.048	0.041
match	13	0.036	0.036
replace	15	0.045	0.045
substring- charAt	10	0.056	0.045
split- startsWith	9	0.066	0.061
lastIndexOf	12	0.036	0.031
lastIndexOf- replace	10	0.042	0.038

6 Conclusion

The paper presents the automated test data generation method based UML sequence diagrams, class diagrams. The method supports UML 2.0 sequence diagrams including all twelve kinds of combined fragments. From CFG, the algorithm for generating test scenarios is developed to avoid test paths explosion without having the points of shared data of threads in parallel or weak sequencing fragments. The constraints of each test scenario and equations at the boundary of variable are converted into input formula of Z3-str solver. Test data are given by a possible model of the Z3-str. Moreover, these test data are satisfiable the constraints with boundary coverage. In test data generation, some preprocessing rules are extended for other operations such as charAt, lastindexOf, trim, startsWith and endsWith. Our tool is implemented to support the proposed method. Some experiments are shown the effectiveness of the tool.

We are also going to develop completely automated test case generation that is automatic standardization the inputs of Z3-str. The proposed method is extended for other UML diagrams (e.g., state-chart diagrams, activity diagrams). Moreover, we would like to investigate, evaluate further the fault–detection effectiveness, costs, and the coverage criteria.

Acknowledgments

This work is supported by the project no. QG.16.31 granted by Vietnam National University, Hanoi (VNU).

References

1. M. Dhineshkumar and Galeebathullah. An approach to generate test cases from sequence diagrams. In: Proceedings of the 2014 International Conference on Intelligent Computing Applications, ICICA '14. IEEE Computer Society, Washington, DC, USA, pp. 345–349 (2014)
2. P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09. ACM, New York, USA, pp.188–198 (2009)
3. M. Shirole and R. Kumar. Testing for concurrency in uml diagrams. SIGSOFT Softw. Eng. Notes, 37(5): pp. 18 (2012)
4. A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09. ACM, New York, NY, USA, pp. 105–116 (2009)
5. A. Nayak and D. Samanta. Automatic test data synthesis using uml sequence diagrams. Journal of Object Technology,vol 9(2): pp. 115–144 (2010)
6. M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
7. M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10. IEEE Computer Society, Washington, DC, USA, pp. 498–507 (2010)
8. T.-D. Vu, P. N. Hung, and V.-H. Nguyen. A method for automated test data generation from sequence diagrams and object constraint language. In: Proceedings of the Sixth International Symposium on Information and Communication Technology, SoICT 2015. ACM, New York, NY, USA, pp. 335-341 (2015)
9. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM New York, NY, USA, pp. 114–124 (2013)

7 Appendix

When analyzing xmi file, parameters of ignore fragment are named parFrag. If fragment is ignore and message m is considered insignificant (line 3), the algorithm makes each message of operand corresponding to BN except for message m. In line 6, if a fragment is consider and a parameter of the fragment is message m, the method only creates a BN corresponding to message m. If assert fragment shows message m1() occur at this point, following by state invariant $\{x == y\}$ (line 9, the algorithm only creates a BN corresponding to m1() if $\{x == y\}$ is true). In line 18, if a fragment is neg and a operand of the fragment is m1 then the algorithm returns exitNode and goes back the Algorithm 1 in [8].

Algorithm 2 Analyzing queue for ignore, consider, neg, assert fragments

Input: Class diagram CD, queue, $curNode \in A$
Output: $exitNode \in A$

```
function processElement(queue, CD:class diagram, curNode:A):A
1: while queue != empty do
2:    x= queue.pop();
3:    if((x==frag)&(x.type=="ignore")&(parFrag==m)& ($operand \neq m$)) then
4:       Create a BN ;
5:       ConnectEdge(curNode,BN);
6:    else if(x==frag)&(x.type=="consider")&(parFrag==m) &(operand==m)then
7:       Create a BN ;
8:       ConnectEdge(curNode,BN);
9:    else if ((x==frag)&(x.type=="assert")) then
10:        Create a DN;
11:        attachGuard_DN; curNode = DN;
12:           if (guard==true) then
13:              Create a BN ;
14:              ConnectEdge(curNode,BN);
15:           end if
16:           Create a MN;
17:           ConnectEdge(curNode,MN);curNode = MN;
18:     else if ((x==frag)&(x.type=="neg")) then
19:        if (operand==m1) then
20:           return exitNode
21:        else
22:           Create a BN;
23:        end if
24:        curNode=BN;
25:     end if
26:     return exitNode;
27: end while;
```