Development of a Tool for Extended Place/Transition Net-Based Mutation Testing and Its Application Example

Tomohiko Takagi
Faculty of Engineering, Kagawa University
2217-20 Hayashi-cho, Takamatsu-shi, Kagawa 761-0396, Japan

Shogo Morimoto
Graduate School of Engineering, Kagawa University
2217-20 Hayashi-cho, Takamatsu-shi, Kagawa 761-0396, Japan

Tetsuro Katayama
Institute of Education and Research for Engineering, University of Miyazaki
1-1 Gakuen-kibanadai nishi, Miyazaki-shi, Miyazaki 889-2192, Japan
E-mail: takagi@eng.kagawa-u.ac.jp, s17g483@stu.kagawa-u.ac.jp, kat@cs.miyazaki-u.ac.jp

Abstract

This paper shows a tool for EPN (Extended Place/transition Net)-based mutation testing to evaluate and improve the quality of a test suite for concurrent software. The tool includes functions for (1) construction of an original EPN that represents the expected behavior of concurrent software under test, (2) construction of mutant EPNs by applying mutation operators to the original EPN, (3) execution of a test suite on each mutant EPN in order to calculate its mutation score, and so on. If the mutation score is not good, the test suite can be improved based on mutant EPNs that have not been killed. The tool was applied to an example of non-trivial software, and it was found that the effectiveness of PN (Place/transition Net)-based mutation testing would be improved by achieving (a) the higher representation power of a PN by the introduction of actions and guards, and (b) the semi-automation by the tool.

Keywords: Software Testing, Mutation Testing, Model-Based Testing, Vienna Development Method

1. Introduction

The techniques to create a test suite (a set of test cases) based on formal models that represent specifications of SUT (Software Under Test) are called MBT (Model-Based Testing),[1] and they can be introduced into a software testing process in order to systematically evaluate and improve the reliability of large SUT. The effects of MBT depend on the quality of a test suite, that is, the capacity of a test suite to find possible failures in SUT. MBMT (Model-Based Mutation Testing) was proposed as a technique to evaluate and improve the quality of a test suite of MBT.

In MBMT, mutant models are created by inserting an intentional failure into a formal model of SUT, and then the ratio of killed mutant models (that is, mutant models whose failures have been found by a test suite) is calculated to evaluate the quality of the test suite. The ratio is called a mutation score, and it can be improved by modifying the test suite based on mutant models that have not been killed.

For example, some studies showed MBMT using directed graphs or automata as their formal model.[2] On the other hand, we have been discussing PNBMT (PN-Based Mutation Testing), that is, MBMT using a PN (Place/transition Net) that can represent concurrent behavior of SUT.[3-5] However, a PN cannot represent actions (data processing that should be executed with state transitions of SUT) and guards (conditions to get transitions fireable). The actions and guards play an important part especially when MBMT (also MBT) is applied to SUT including complex data processing.

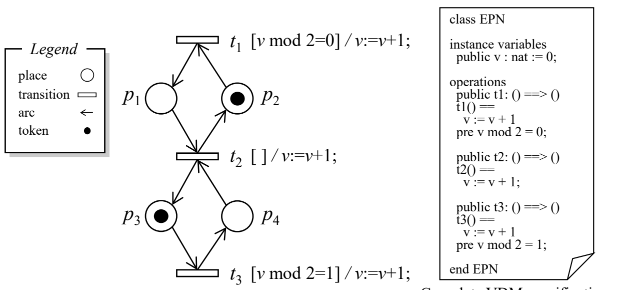
In this paper, we propose an EPN (Extended Place/transition Net), that is, a PN that is extended by introducing VDM++ (a formal specification description language for VDM (Vienna Development Method))[6] in order to define actions and guards, and then, we show a tool that we have been developing to support EPNBMT (EPN-Based Mutation Testing). The tool is supposed to be used by test engineers. This study is based on the idea that the effectiveness of PNBMT will be improved by achieving (a) the higher representation power of a PN by the introduction of actions and guards, and (b) the semi-automation by our tool.

The rest of this paper is organized as follows. In section 2, we propose an EPN. Section 3 shows the principal functions of our tool according to the procedure of EPNBMT. Section 4 then shows an application example, and section 5 describes related work. Finally, section 6 gives a conclusion and future work.

2. EPN (Extended Place/transition Net)

In this section, we propose an EPN, that is, a PN that is extended by introducing VDM++ (a formal specification description language for VDM) in order to define actions and guards.

An EPN consists of a PN and VDM specifications. The PN is a traditional formal model suitable for representing expected state transitions of SUT including concurrent behavior. It consists of places to represent the states of components of SUT, transitions and arcs to represent the change of states of SUT, and tokens to represent current states or resources of the components. The VDM specifications are a set of VDM++ codes to represent actions and guards that are given to transitions of the PN if necessary. The actions can represent data processing that should be executed with state transitions



```
class EPN

instance variables
  public v : nat := 0;

operations
  public t1: () ==> ()
  t1() ==
    v := v + 1
  pre v mod 2 = 0;

  public t2: () ==> ()
  t2() ==
    v := v + 1;

  public t3: () ==> ()
  t3() ==
    v := v + 1
  pre v mod 2 = 1;

end EPN
```

Complete VDM specifications

Fig. 1. Simple example of an EPN (initial state).

of SUT, and the guards can represent conditions to get transitions fireable.

In an EPN, a state of SUT corresponds to a set that consists of a marking (that is, a distribution of tokens on a PN) and values of instance variables (that is, variables that are defined in VDM specifications and are used in actions and guards), and a state transition of SUT corresponds to the change of a marking and values of instance variables that results from the firing of a transition of a PN.

A simple example of an EPN is shown in Fig. 1. The initial marking of the EPN is expressed as [0,1,1,0], since the numbers of tokens in the places p_1, p_2, p_3, p_4 are 0, 1, 1, 0, respectively. Also, the instance variable v is initialized to 0, and therefore the initial state of the EPN is expressed as {[0,1,1,0], v=0}. The action "v:=v+1;" is attached to the transitions t_1, t_2, t_3, and it is executed by the firing of each transition. Additionally, the guards "v mod 2=0" and "v mod 2=1" are attached to t_1 and t_3, respectively. t_1 and t_3 can be fired, when there are tokens for their ingoing arcs and their guards are satisfied. For example, only t_1 is fireable in the initial state of the EPN of Fig. 1.

3. Tool for EPNBMT (Extended Place/transition Net-Based Mutation Testing)

This section shows the functions of the tool that we have been developing to support EPNBMT, according to the procedure of EPNBMT. The functions (the procedure) consist of (1) construction of an original EPN, (2) construction of mutant EPNs, (3) conversion of a test suite, and (4) execution of a test suite.

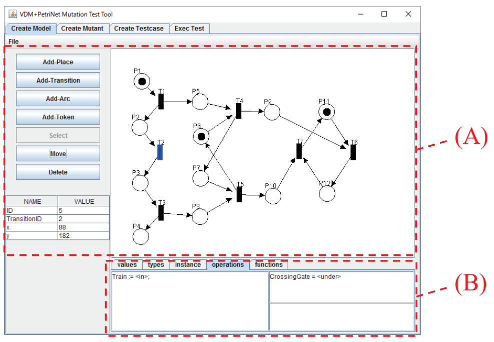


Fig. 2. Screen image of the tool in constructing an original EPN.

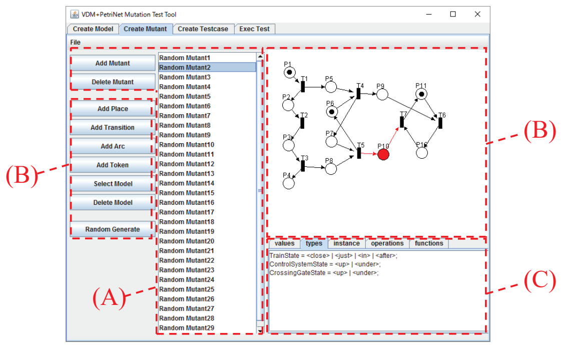


Fig. 3. Screen image of the tool in constructing mutant EPNs.

3.1. *Construction of an original EPN*

First, test engineers need to construct an original EPN based on specifications of SUT. The original EPN is an EPN that represents the expected behavior of SUT; it is called "original" in order to distinguish it from mutant EPNs that will be constructed on the next step. The EPN has been discussed in the previous section.

Our tool provides the function to construct an original EPN. Fig. 2 gives the screen image in the execution of the function. The GUI (Graphical User Interface) for the function consists of (A) a pane to edit a PN and (B) a pane to edit VDM specifications for actions and guards. Additionally, (B) consists of five code editors to edit values, types, instance variables, operations, and functions of the VDM specifications, and one of them can be selected by a tab. Test engineers can define not only the PN of an original EPN but also the actions and guards of an original EPN.

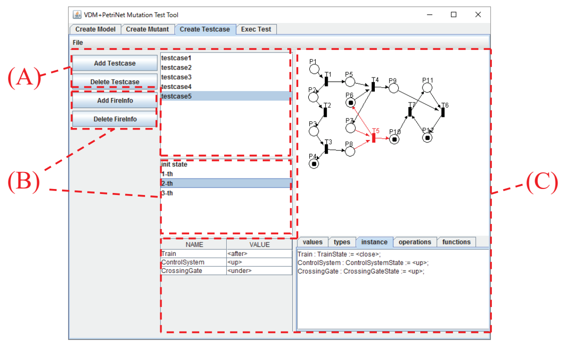
3.2. *Construction of mutant EPNs*

After completing constructing an original EPN, test engineers need to construct mutant EPNs. The mutant EPN is an EPN that intentional failure(s) has been inserted into by the use of mutation operators. In this study, the following mutation operators are supposed to be utilized.

- Model-based mutation operators: the insertion and omission of an element of a PN (that is, a place, a transition, an arc, or a token) can be performed in order to insert an intentional failure. They were proposed in our previous study.[3]
- Code-based mutation operators: the grammar of VDM++ is somewhat similar to those of general procedural programming languages. Therefore, in order to insert an intentional failure, traditional code-based mutation operators (for example, replacement of operators, variables, and constants) can be applied to actions and guards that have been implemented as VDM specifications.

Our tool provides the function to construct mutant EPNs. Fig. 3 shows the screen image in the execution of the function. The GUI for the function consists of (A) a list of mutant EPNs that have been constructed, (B) a pane to edit a PN of a mutant EPN by the use of model-based mutation operators, and (C) a pane to edit VDM specifications for actions and guards by the use of code-based mutation operators.

(A) enables test engineers to select one of existing mutant EPNs and show it on (B) and (C). Additionally, it enables to add and remove a mutant EPN. (B) highlights a part that a mutation operator has been applied to in order that test engineers can easily confirm a result of application of a mutation operator. (B) also gives test engineers the way of manual and automatic construction of a mutant EPN. When performing manual construction, test engineers select a model-based mutation operator from the menu, and then specify a part of a PN that it should be applied to. When executing automatic construction, a model-based mutation operator is randomly selected, and then a part that it will be applied to is randomly determined by our tool. (C) consists of five code editors to edit values, types, instance variables, operations, and functions of VDM specifications, and one of them can be selected by



testcase1
testcase2
testcase3
testcase4
testcase5

init state
1-th
2-th
3-th

NAME	VALUE
Train	<after>
ControlSystem	<up>
CrossingGate	<under>

```
Train : TrainState := <close>;
ControlSystem : ControlSystemState := <up>;
CrossingGate : CrossingGateState := <up>;
```

Fig. 4. Screen image of the tool in converting a test suite.

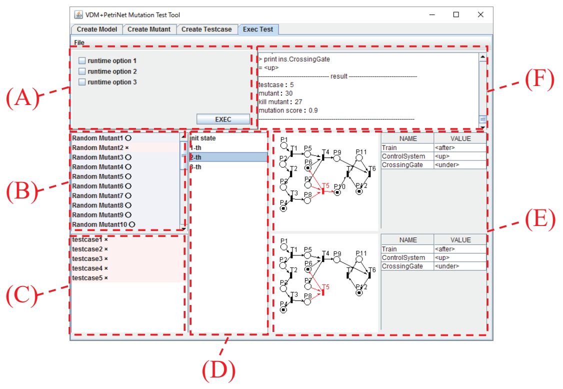
a tab. These code editors enable test engineers to manually apply a code-based mutation operator.

3.3. *Conversion of a test suite*

After the construction of an original EPN, a test suite to be evaluated needs to be converted to execution paths (that is, sequences of successive markings, transitions, and values of instance variables) on the original EPN. It is a preparation for automatic execution of the test suite in the next step. Our EPNBMT accepts an arbitrary format and design technique of the test suite.

Our tool provides the function to manually convert an arbitrary test suite to execution paths on an original EPN. Fig. 4 shows the screen image in the execution of the function. The GUI for the function consists of (A) a list of test cases that a test suite consists of, (B) a list to edit transitions of a test case selected on (A), and (C) a pane to show an original EPN and edit a test case.

(A) enables test engineers to select one of existing (already converted) test cases and show its execution path representation on (B) and (C). (C) highlights a transition that is selected on (B), and shows a marking and values of instance variables just after the firing of the transition. Additionally, (A) enables to add and remove a test case. When adding a test case (converting a test case and entering it into our tool), test engineers input required information (successive markings, transitions, and values of instance variables) via (B) and (C). (B) enables to add and remove a transition of a test case selected on (A). When a new transition is added, (C) can highlight fireable transitions based on a current marking and values of instance variables, and then



```
> print ins.CrossingGate
= <up>
------------------------------ result ------------------------------
testcase : 5
mutant : 30
kill mutant : 27
mutation score : 0.9
--------------------------------------------------------------------
```

Random Mutant1 ○
Random Mutant2 ×
Random Mutant3 ○
Random Mutant4 ○
Random Mutant5 ○
Random Mutant6 ○
Random Mutant7 ○
Random Mutant8 ○
Random Mutant9 ○
Random Mutant10 ○

testcase1 ×
testcase2 ×
testcase3 ×
testcase4 ×
testcase5 ×

init state
1-th
2-th
3-th

NAME	VALUE
Train	<after>
ControlSystem	<up>
CrossingGate	<under>

NAME	VALUE
Train	<after>
ControlSystem	<up>
CrossingGate	<under>

Fig. 5. Screen image of the tool in executing a test suite.

selecting a highlighted transition brings the automatic updating of the current marking and values of instance variables, which will save test engineers' labor.

3.4. *Execution of a test suite*

Finally, test engineers need to apply the converted test suite to all of the constructed mutant EPNs, and then calculate a mutation score as the ratio of killed mutant EPNs to all of the constructed mutant EPNs. If the mutation score is not good, the test suite can be improved based on mutant EPNs that have not been killed.

Our tool provides the function to execute a test suite. Fig. 5 shows the screen image in the execution of the function. The GUI for the function consists of (A) a control panel to start executing a test suite, (B) a list of mutant EPNs with the results of testing, (C) a list of test cases with the results of their application to a mutant EPN selected on (B), (D) a list of transitions of a test case selected on (C), (E) a pane to show markings and values of instance variables of an original EPN (the upper part) and the selected mutant EPN (the lower part) just after the firing of the selected transition in the selected test case, and (F) a pane to show a detailed log of test suite execution, including a mutation score. (B)-(F) enable test engineers to easily confirm the result of test suite execution invoked by (A).

Each test case is executed as follows:

(i) A mutant EPN is initialized based on an initial marking and initial values of instance variables of a test case.

(ii) In the mutant EPN, transitions are fired from the top of the test case. Each of firings is performed by the following (a)-(c).
  (a) The pre-conditions to fire a transition, that is, the existence of tokens for ingoing arcs of the transition, and the guard of the transition are checked. When they are not satisfied, it is found that the mutant EPN is killed by the test case.
  (b) The transition is fired, and the action attached to the transition is executed. As a result, a current marking and values of instance variables are updated.
  (c) If the transition includes post-conditions, and they are not satisfied, it is found that the mutant EPN is killed by the test case.

(iii) When the firing of the final transition of the test case has been completed, the test result (that is, a final marking and values of instance variables in the mutant EPN) is compared with the expected result included in the test case (that is, a final marking and values of instance variables in the original EPN). If there are differences between the test result and the expected result, it is found that the mutant EPN is killed by the test case.

4. Experiment

In order to evaluate the effectiveness of EPNs (discussed in section 2) and the tool (discussed in section 3), we performed an experiment using part of an OFMS (Online File Management System).

The OFMS is an example of non-trivial software that was given in Ref. 7 in order to discuss MBT using PNs. The PN that represents the abstracted expected behavior of the OFMS was constructed in Ref. 7. However, it did not include formal definitions of detailed expected behavior based on complex data processing and conditions, due to the limitation of representation power of PNs. To address this problem, we constructed an EPN of the OFMS, that is, we added actions and guards to the reconstructed PN by the use of the function of the tool shown in section 3.1. The EPN consists of 7 places, 17 transitions, 32 arcs and 35 lines of VDM++ code, and it properly represents the above-mentioned detailed expected behavior. The effort spent in constructing the EPN is about 6 man-hours. It is found that EPNs have higher representation power than PNs, and are useful to define the detailed expected behavior of SUT by reasonable effort.

We constructed mutant EPNs from the EPN of the OFMS (that is, original EPN) by the use of the function of the tool shown in section 3.2. The tool generated 25 mutant EPNs by the automated application of model-based mutation operators in about 3.5 milliseconds. The environment in which the tool was executed is a personal computer with i5-6500T processor (2.50 GHz, up to 3.10 GHz) and 4 GB RAM. Test engineers can cut down on the effort to apply model-based mutation operators. Additionally, 25 mutant EPNs were created by the manual application of code-based mutation operators in about 1.5 man-hours. The code-based mutation operators to insert intentional failures into actions and guards cannot be used in previous PNBMT, since PNs do not include actions and guards.

We then created a test suite to be evaluated, by the use of the function of the tool shown in section 3.3. The test suite covers the transitions of the original EPN. Covering the elements of a model is a testing strategy that is in general use.

Last of all, we applied the test suite to all of the mutant EPNs and got a mutation score by the use of the function of the tool shown in section 3.4. This step is automatically executed by the tool, and therefore test engineers can cut down on effort. The execution by the tool was completed in about 208 seconds, which is short enough. The mutation score of the test suite was 0.72. We analyzed the mutant EPNs that had not been killed, and it was found that some codes of actions and guards (for example, processing of the constraint on a storage capacity) had not been tested. Actions and guards cannot be analyzed in previous PNBMT, and therefore the quality of a test suite would be evaluated and improved more effectively in EPNBMT.

The conclusion of this experiment is that, as mentioned in section 1, the effectiveness of PNBMT can be improved by achieving (a) the higher representation power of a PN by the introduction of actions and guards, and (b) the semi-automation by our tool. However, there is still room for improvement, which is discussed at the end of section 6 as future work.

5. Related Work

This section describes related work on the area of MBT and MBMT using a PN and its similar models.

A PN is a kind of Petri net, and MBT using Petri nets is known as a technique effective for systematically testing SUT including concurrent behavior. For example, Ref. 8 shows a technique to construct a HPN (High-level Petri Net) that represents the workflow of BPEL-based Web service composition and to test using the HPN. In the technique that was proposed in Ref. 9, a time Petri net that represents the behavior of real-time software is decomposed into independent segment groups, and test cases based on a timing criterion are generated using them. Also, Ref. 7 gives a technique in which test cases that cover the state transition sequences of specified length are generated from a PN that represents the behavior of software. Note that actions were introduced into a PN in Ref. 7, but they are different from the actions of an EPN. The action of a PN in Ref. 7 is the definition of detailed test procedures (that is, tasks to be performed by test engineers or a test execution tool), and it is used to generate test instructions for the test engineers or test scripts for the test execution tool.

On the other hand, MBMT is relatively new research area. Ref. 2 shows insertion and omission as basic mutation operators for graph-based models. Based on the idea that model-based mutation operators can be defined as (the combination of) insertion and omission of model elements, we proposed eight kinds of mutation operators for PNs in Ref. 3, and then we introduced them into EPNBMT. Ref. 4 shows a framework for both of mutation analysis and negative testing in PNBMT. Part of the framework was implemented and extended in our tool shown in this paper. In Ref. 5, a technique to calculate an extended mutation score based on the weights of a PN was proposed to evaluate a test suite accurately. The weights are calculated based on software metrics that reflect fault-proneness and the impact on reliability. The extended mutation score would be useful for the improvement of EPNBMT as well as PNBMT.

6. Conclusion and Future Work

This paper has shown a tool for EPNBMT (Extended Place/transition Net-Based Mutation Testing) to evaluate and improve the quality of a test suite for software including concurrent behavior. The EPN proposed in this paper is a PN extended by introducing a formal specification description language for VDM, in order to represent actions and guards. The tool consists of the following four functions: (1) an original EPN can be created to define the expected behavior of SUT, (2) mutant EPNs including intended failures can be created by applying mutation operators to the original EPN, (3) an arbitrary test suite to be evaluated can be converted to execution paths on the original EPN, and (4) a mutation score can be calculated by applying the converted test suite to all the mutant EPNs. If the mutation score is not good, the test suite can be improved based on mutant EPNs that have not been killed. The tool was applied to an example of non-trivial software, and it was found that the effectiveness of PNBMT would be improved by achieving (a) the higher representation power of a PN by the introduction of actions and guards, and (b) the semi-automation by the tool.

In the tool, model-based mutation operators are automatically applicable, but code-based mutation operators are only for manual application. The latter will be automated in order to save test engineers' labor in our future work. Additionally, if the tool can generate mutant EPNs intensively including failures that are possible in operational environments, a degree of precision of a mutation score would be improved. Thus we will plan to develop the techniques for modeling of actual failures and for strategical application of mutation operators based on the model.

Another approach to improve the degree of precision of a mutation score is to utilize fault-proneness information that can be derived from software metrics, in order to adjust a mutation score. In the previous study, we proposed a technique in which the weights based on fault-proneness information and usage distributions are given to a PN in order to calculate an adjusted mutation score[5]. Extending this technique for EPNs is also included in our future work.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number 26730038.

References

1. M. Utting, A. Pretschner and B. Legeard, A taxonomy of model-based testing approaches, *Software Testing, Verification and Reliability*, Vol.22 (2012), pp.297-312.
2. F. Belli, C.J. Budnik and E. Wong, Basic Operations for Generating Behavioral Mutants, in *Proc. of 2nd Workshop on Mutation Analysis in conjunction with ISSRE'06* (Nov. 2006), pp.9.
3. T. Takagi, R. Takata, Z. Furukawa, F. Belli and M. Beyazıt, Metrics for Model-Based Mutation Testing Based on Place/Transition Nets, in *Proc. of Joint Conf. of 21st Int. Workshop on Software Measurement and 6th Int. Conf. on Software Process and Product Measurement (IWSM-MENSURA)* (Nov. 2011), pp.7-10.
4. T. Takagi and T. Arao, Overview of a Place/Transition Net-Based Mutation Testing Framework to Obtain Test Cases Effective for Concurrent Software, in *Proc. of 16th Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (June 2015), 3 pages.
5. T. Takagi and T. Teramoto, Extended Mutation Score Based on Weighted Place/Transition Nets to Evaluate Test Suites, in *Proc. of 15th Int. Conf. on Computer and Information Science (ICIS)* (June 2016), pp.959-961.
6. J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat and M. Verhoef, *Validated Designs for Object-Oriented Systems*, (Springer-Verlag London, 2005).
7. T. Takagi and Z. Furukawa, Test Case Generation Technique Based on Extended Coverability Trees, in *Proc. of 13th Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (Aug. 2012), pp.301-306.
8. W. Dong, H. YU and Y. Zhang, Testing BPEL-based Web Service Composition Using High-level Petri Nets, in *Proc. of 10th International Enterprise Distributed Object Computing Conference (EDOC)* (Oct. 2006), pp.441-444.
9. I. Ho and J. Lin, Generating Test Cases for Real-Time Software by Time Petri Nets Model, in *Proc. of 8th Asian Test Symposium (ATS)* (Nov. 1999), pp.295-300.