Journal of Robotics, Networking and Artificial Life, Vol. 1, No. 3 (December 2014), 207-211

Proposal of a Supporting Method for Debugging to Reproduce Java Multi-threaded Programs by Petri-net

Tetsuro Katayama[*†], Shoichiro Kitano[†], Yoshihiro Kita[‡], Hisaaki Yamaba[†] and Naonobu Okazaki[†]
[†]*Faculty of Engineering, University of Miyazaki, Miyazaki 889-2192, Japan*
[‡]*Security Center, Kanagawa Institute Technology, Kanagawa 243-0292, Japan*

[*]Corresponding author, E-mail: kat@cs.miyazaki-u.ac.jp
Tel: +81-985-58-7586, Fax: +81-985-58-7586

Abstract

It is difficult to implement the multi-threaded programs. The reason is that the behavior of each thread is non-deterministic. Also it is difficult to reproduce the situation in which an incident occurs. This paper proposes a supporting method for debugging to reproduce Java multi-threaded programs by visualizing the behavior of the programs with Petri-net. Moreover, we have confirmed the effectiveness of our method by implementing a tool.

Keywords: multi-threaded program, debugging, Petri-net, Java, reproducibility

1. Introduction

In recent year, many computers are adopted multi-core CPUs. For using such resources effectively, the demand of multi-threaded programs increases.

In multi-threaded programs, it is hard work for even expert programmers to implement them, and easier to embed bugs than single-threaded programs[1]. Most of such bugs are discovered in latter half of the development process or in executing the programs by users. Therefore, it is difficult to fix the programs. It is necessary to remove the bugs at the unit testing to resolve this problem.

One of the testing methods executes a program with plural interleaving by putting off a timing of execution of each thread. Hereby, we can discover potential bugs in a multi-threaded program.

However, even if this testing method shows that bugs exist, in multi-threaded programs, it is hard to discover the cause of the bugs in debugging because the behavior of multi-threaded programs is non-deterministic.

This paper proposes a supporting method for debugging to reproduce multi-threaded programs by Petri-net to improve efficiency of debugging work for the multi-threaded programs written in Java language.

Specifically, the proposed method gives reproducibility to multi-threaded programs, generates data file for execution path of multi-threaded program, and simulates the behavior of the program by Petri-net based on the data file.

Here, ordinal Petri-net cannot express the behavior of multi-threaded programs written in Java completely.

2. Supporting Method Using Petri-net

2.1. *Supporting method*

This section proposes a supporting method for debugging that improves the efficiency of discovering the cause of bugs by giving reproducibility to multi-thread programs. The procedure is following.

(i) Generating the data file of the execution path in the situation in which an incident occurs
The data has ID of threads executing process, executed processes, timing of processes, and ID of generated threads.

Published by Atlantis Press

```
public class Example {

    static class ThreadExample extends Thread {
        public void run() {
            Object lock = new Object();
            synchronized (lock) {
                System.out.println("in synch");
            }
            System.out.println("lock release");
        }
    }

    public static void main(String[] args) {
        System.out.println("Hello");
        new ThreadExample().start();
        System.out.println("World");
    }
}
```

Fig. 1. Example of a Multi-threaded Program written in Java.

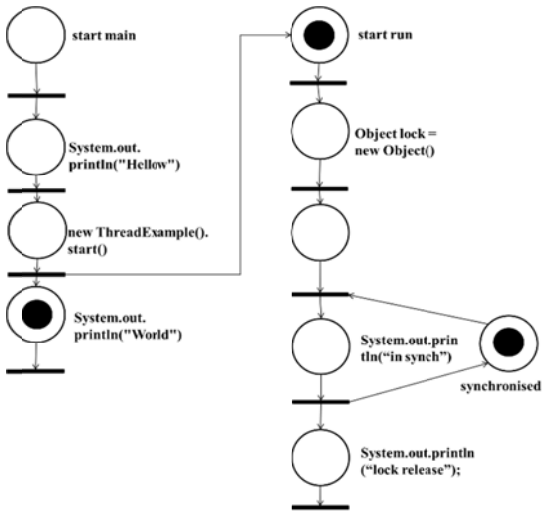

Fig. 2. Petri-net Modeling of Fig.1.

(ii) Generating a Petri-net model from a tested Java program
We explain the method to model Java programs by Petri-net in the next section.

(iii) Simulating the behavior of Java program which has tested
This process uses the data file and a Petri-net model.

2.2. *Modeling Petri-net for Java programs*

This paper models Java programs by Petri-net using the conversion rule as follows. Fig.2 is an example of a Petri-net model for the program of Fig.1 which is an example of a multi-threaded program written in Java.

- a statement is converted into a place,
- a reserved word 'synchronized' is converted into a place,
- a state of waiting lock is converted into a place,
- a beginning of the method is converted into a place,
- a transition to the next statement is converted into a transition,
- a thread is converted into a token,
- a locked instance is converted into a token.

Fig. 3. Model for Identifying a Thread and a Locked Instance.

2.3. *Extending Petri-net*

To improve the efficiency of discovering the cause of bugs, we extend Petri-net. This extension can support to understand the behavior of Java programs more intuitively.

2.3.1 *Identifying tokens by ID*

Because of inability of Petri-net to identify the each token which passes as the same path, we enable Petri-net to identify each token by adding ID to tokens.

2.3.2 *Expressing the role of tokens by color*

The modeling rule which we have described in section II cannot make us to understand that the token expresses thread or locked instance intuitively. Thus, we define a token expressing thread is a white circle and a locked instance is a black circle as shown in Fig.3. This definition makes us easy to understand the role of tokens.

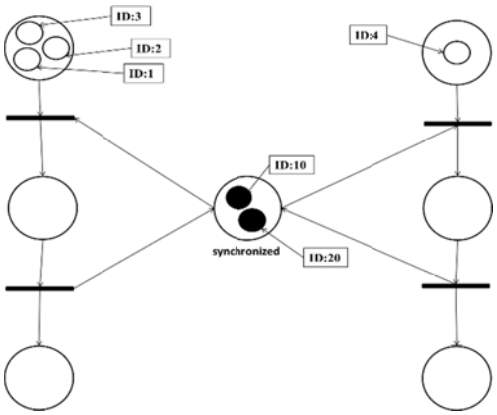

Fig. 4. Example of Petri-net with ID.

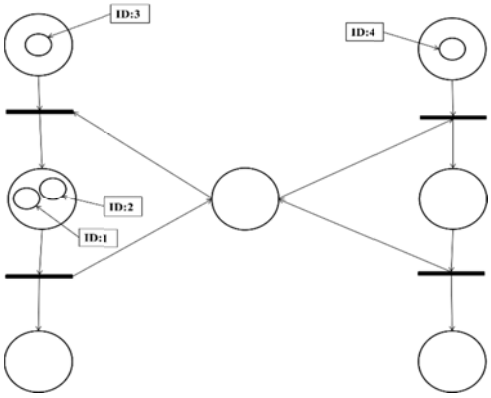

Fig. 5. State of Waiting for a Locked Instance.

2.3.3 *Expressing locked instances*

At present, it is hard to understand our Petri-net when many locked instances exist in the model or many threads share one locked instance. Also, there is a problem not to know which thread locks which instance.

As an example of these problems, Fig.4 expresses that there are threads numbered ID 1,3,4 which use the locked instance numbered ID 10 and thread numbered ID 2 which use the locked instance numbered ID 20. However, such information is not described in the Petri-net model. As shown in Fig.5, it is impossible to confirm the cause of waiting threads with ID3,4 when the thread with ID1,2 have moved to the next place.

Hence, we extend a token expressing a thread to describe ID which should be locked by each thread. Furthermore, in order to express the state of a thread locking an instance, we extend a token expressing a thread to describe the locked token on the thread token. This extension can express the state of releasing the lock in synchronized block happened by wait() method which is a method of java.lang.Object class.

3. Confirmation

We confirm the effectiveness of proposed supporting method for debugging by implementing a tool.

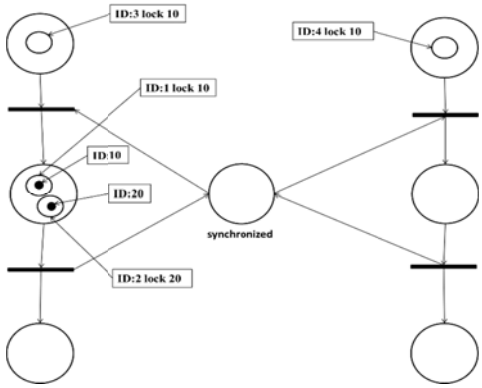

Fig. 6. Extended Petri-net for Java.

3.1. *The method of confirmation*

As an example, we use a multi-threaded program without the process of synchronization written in Java. Fig.7 shows the code of the example.

This program outputs strings by three classes succeeding java.lang.Thread class. FirstNamePrinter class outputs “Shoichiro”, SpacePrinter class outputs “ ”, and SurnamePrinter outputs “Kitano”. The expected result is a string “Shoichiro Kitano”. However, because of this program without process of synchronization, the result is often different from the expected result.

Fig.8 shows a part of the code generated by our tool from the tested code. An unexpected result is discovered by executing the code. Furthermore, our tool can generate the data file of an execution path in testing the code.

```
public class NakedNamePrinter {

    private final String firstName;
    private final String surName;

    public NakedNamePrinter(String firstName, String surName) {
        this.firstName = firstName; //NakedNamePrinter 1
        this.surName = surName; //NakedNamePrinter 2
        new FirstNamePrinter().start(); //NakedNamePrinter 3
        new SpacePrinter().start(); //NakedNamePrinter 4
        new SurnamePrinter().start(); //NakedNamePrinter 5
    }

    private class FirstNamePrinter extends Thread {
        public void run(){
            System.out.print(firstName); //FirstNamePrinter 1
        }
    }

    private class SpacePrinter extends Thread {
        public void run() {
            System.out.print(' '); //SpacePrinter 1
        }
    }

    private class SurnamePrinter extends Thread {
        public void run() {
            System.out.println(surName); //SurnamePrinter 1
        }
    }

    public static void main(String[] args) {
        new NakedNamePrinter("Shoichiro", "Kitano"); //main 1
    }
}
```

Fig. 7. A Tested Code.

By using a data file generated by executing the code shown in Fig.8, we have confirmed the effectiveness of our proposed method by letting students in our laboratory use our tool.

3.2. *Result*

Fig.9 shows Petri-net generated by our tool. In Fig.9, the label with number added to the place corresponds with the comment in Fig.7, and the place expresses statement with corresponded the comment.

Fig.10 shows the reproduction of situation when an incident occurs. Primarily, the place should be marked in order of “FirstNamePrinter 1”, “SpacePrinter 1”, and “SurnamePrinter 1” in this Petri-net. In Fig.10, Because of a token is on “SpacePrinter start run”, it realizes that “SpacePrinter 1” is not executed and “SurnamePrinter 1” has been executed earlier than “SpacePrinter 1” by the marking of a place with “SurnamePrinter 1”.

Here, we can understand that threads are not executed in order our expectation. Thus, we have noticed that there is a defect in synchronized process between threads.

```
public class NakedNamePrinter {

    private final String firstName;
    private final String surName;

    public NakedNamePrinter(String firstName, String surName) {
        long t;
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, “start  NakedNamePrinter");

        this.firstName = firstName;
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, "this.firstName = firstName");

        this.surName = surName;
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, "this.surName = surName");

        new FirstNamePrinter().start();
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, "new FirstNamePrinter().start()");

        new SpacePrinter().start();
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, "new SpacePrinter().start()");

        new SurnamePrinter().start();
        t = System.nanoTime();
        ExecutionWriter.addExecutionData(t, "new SurnamePrinter().start()");
    }
```

Fig. 8. A Part of the Generated Code.

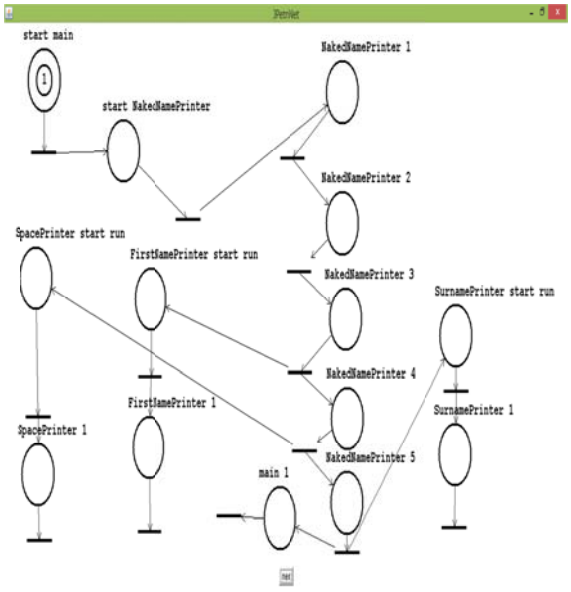

Fig. 9. Petri-net Generated by Tool.

From this, because of reproducing the situation in which an incident occurs and understanding the behavior of a program graphically, it is easy to discover a cause of a bug. Thus, we have confirmed our method is effectively.

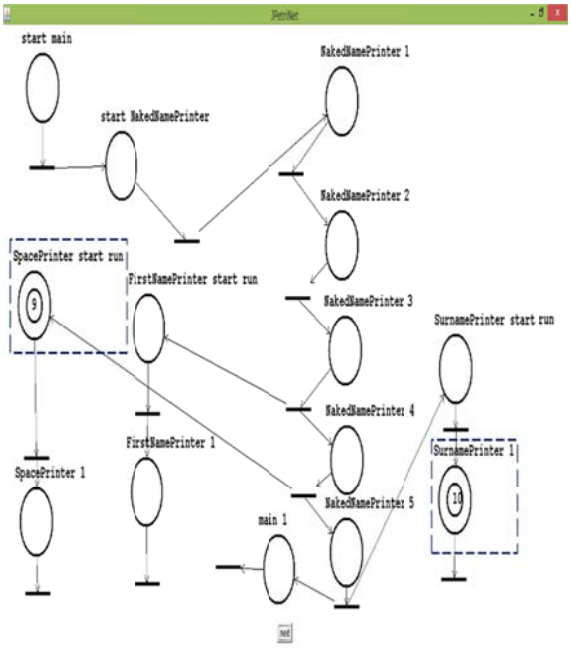

Fig. 10. Situation in Which an Incident Occurs.

3.3. *Comparison with related work and existing tool*

As related work, some researches discover potential bugs of multi-threaded programs by automatically generated Petri-net models[2][3][4]. In these researches, it is possible to discover the existence of bugs by analyzing Petri-net statically. However, this approach cannot discover the cause of bugs.

There is a tool for multi-threaded programs that discover potential bugs by executing the unit testing automatically[5]. This tool cannot reproduce the situation in which an incident occurs to test the multithreaded program by plural interleaving at random. Therefore, discovering cause of bugs is depended on users.

On the other hand, our debugging method can reproduce the same situation in many times, because our method uses the data file of an execution path that was really executed. It makes us easy to discover the cause of bugs. Therefore, it may be said that our proposing method is effective for removing the potential bug of the multi-thread program.

4. Conclusion

In this paper, we have proposed supporting method for debugging to reproduce Java multi-threaded programs by Petri-net. And we have extended Petri-net to understand the behavior of Java programs more. We think that this extension improves the efficiency of debugging. Furthermore, we have shown the effectiveness of our method by adapting a tool to an example and comparing with related works and the existing tool.

Future issues are as follows.

- Improving of the adaptation range of our tool,
- Confirming effectiveness of extended Petri-net.

References

1. J. K. Ousterhout, Why Threads Are A Bad Idea (for most purposes), Presentation given at the 1996 Usenix Annual Technical Conference, (1996), http://www.softpanorama.org/People/Ousterhout/Threads
2. K. M. Kavi, A. Moshtaghi, D. Chen, Modeling Multithreaded Applications Using Petri Nets International Journal of Parallel Programming, Vol. 30, No. 5, (2002), pp. 353-371.
3. G.R. Suciu, F. Zuberek, W.M, Timed Petri net Models of Multithreaded Multiprocessor Architectures, IEEE Proceedings of the Seventh International Workshop on Petri Nets and Performance Models, (1997), pp. 153-162.
4. H. Liao, Y. Wang, H. K. Cho, J. Stanley, T. Kelly, S. Lafortune, S. Mahlke, S. Reveliotis, Concurrency bugs in multithreaded software, modeling and analysis using Petri nets, Discrete Event Dynamic Systems, (2013), Vol. 23, Issue 2, pp. 157-195
5. ConTest - A Tool for Testing Multi-threaded Java Applications, https://www.research.ibm.com/haifa/projects/verification/contest/