A Method for Detecting Unusual Defects in Enterprise System Using Model Checking Techniques

YOSHITAKA AOKI
Graduate School of Engineering, Shibaura Institute Technology
Course of Electronic Engineering and Computer Science
Saitama, Japan
m109002@shibaura-it.ac.jp

SAEKO MATSUURA
Graduate School of Engineering, Shibaura Institute Technology
Institute of Technology Course of Electronic Engineering and Computer Science
Saitama, Japan
matsuura@se.shibaura-it.ac.jp

Abstract: - This paper proposes a method based on model checking for detecting hard-to-discover defects in enterprise systems. Source codes are transformed into an appropriate phased abstract model so that we can observe the phenomena. UPPAAL, which is a typical model checking tool, makes an exhaustive checking of the model and provides a result whether the model can reach the specified state or not. We have developed a supporting tool to narrow the range of model checking and to generate UPPAAL model automatically. We discuss our method in detail on the basis of the results of a case study.

Key-Words: - Model checking, bugs, UPPAAL, Infinity Loop, Abstraction, Enterprise System

1 Introduction

Generally, enterprise systems consist of functions based on typical business logic. Business rules include many types of functions, and there are numerous combinations of execution orders and the conditions under which these orders must be executed. Programmers often have inadequate knowledge of business rules and hence may misunderstand the system designer’s intentions. Moreover, programmers occasionally define the incorrect order in which business rules must be applied and the specific conditions under which these rules must be executed. Because some programmers’ decisions are incongruous with the designer’s intentions, it becomes difficult to detect defects arising from the inappropriate definition of business rules by reviewing and testing of the code alone. Generally, not all of inspectors know the whole of source codes well. Ko et al.[1] argued that the inspector generally can take aim at a part of codes including defects by using a file name which represents a brief outline of source codes. Moreover, he/she selects a part of source codes by a doubtful file name and checks them to find out the cause of defects. When the expected defects can not be found, the above-mentioned task is repeated until they are found. Thus, detection of such defects becomes a laborious task even if the inspector has the same knowledge as the system designer. Recently, a model-checking technique that facilitates thorough verification of every possible combination of states and thus ensures that the system reaches the desired state has been proposed as a promising approach for preliminary system development.

This paper proposes a method based on model checking for detecting defects in the Java source code. Source codes are transformed into an appropriate phased abstract model so that we can observe the phenomena. UPPAAL[2], which is a typical model checking tool, makes an exhaustive checking of the model and provides a result whether the model can reach the specified state or not. We have developed a supporting tool to narrow the range of model checking and to generate UPPAAL model automatically. A supporting tool that allows the inspector to detect errors in the manner in which business rules and the required conditions are combined and confirm whether the designer’s intentions have been honored has also been developed. The tool translates the target source code into a model that can be checked by the model-checking tool UPPAAL.

The remainder of the paper is organized as follows. Section 2 discusses the UPPAAL model on the basis of a comparison with other model-checking tools. Section 3 discusses how defect source codes are translated into the UPPAAL model. Section 4 describes the inspection process and Section 5 describes a case study. Section 6 discusses the effectiveness of our method, and Section 7 discusses related work.
2 Model-Checking Tools

Model checking has been favored as a technique that helps improve reliability in the upper process of software development. Model-checking tool uses temporal logic. In model checking, one verifies whether or not the system reaches a specified state. When the specified state is not filled, the tool presents a counter-example. The process can be confirmed by the simulation facility. There are several model-checking tools, including SPIN[3] and PathFinder[4]. Table 1 presents a comparison of these tools with UPPAAL.

Enterprise system developers generally have little knowledge of model-checking techniques. As Table 1 shows, UPPAAL only provides a graphical visualization and recording of the dynamic behavior of a system description, so that we can confirm sequences of symbolic states of the system. It is easy to intuitively understand the phenomenon because UPPAAL can visually confirm laboratory results. As UPPAAL can accommodate time restrictions, it has the advantage of the modeling and verification abilities of real-time embedded systems. Furthermore, UPPAAL is easier to translate from not only Java but also other programming languages.

We adopted the model-checking tool UPPAAL because of the graphical view and fine simulation function in which counter-examples can be easily confirmed.

![Figure1. Basic Components of the UPPAAL Model](image)

3 Modeling the Source Code

3.1 Modeling Difficulties

To detect problems associated with the manner in which the execution orders and the necessary conditions are combined, the source code, including the defects, must be correctly translated into an UPPAAL model. If every point of finishing executing an expression in the source code translates into a location, there is a possibility of the number of states in the system increasing dramatically. As a result, testing may or may not be completed in time. However, the phenomena causing the defects must be correctly reproduced and the cause specified. Defining the cause of these defects enables specification of a sequence of transition arrows and the required conditions.

We must also devise a method for translating the source codes into UPPAAL models so as to identify the root cause of the defects.

3.2 Modeling Policy

To avoid state space explosion, source codes should be translated into an appropriate abstract UPPAAL model to detect the anticipated defects. Generally, we intend to detect phenomena such as infinite loops, data overflow, time-out, and dead-lock problems. These phenomena are caused when the designers’ intentions are not followed or when an incorrect combination of execution orders and the required conditions is employed.

As Ko, et al. said in their paper, it is not always that we can accomplish a task which creates a suitable abstract model so as to detect the specified phenomena at our first try. Therefore, we define an initial abstract model and refine it in the following phased procedure.

Procedure 1:
To observe phenomena caused by the defects, an initial model is defined by extracting characteristic states of each phenomenon from the source codes.
Defects in enterprise system are caused by numerous combinations of execution orders and the conditions under which these orders must be executed. So, a control flow model needs to be extracted from source codes in accordance with "if" statements and "for" statements as mentioned in Section 3.3 and 3.4. In case of such a phenomenon as infinite loops, a state where an infinite loop occurs needs to be added to the initial model.

Procedure 2: Model checking tool UPPAAL verifies that a model can hold a given property. A phenomenon as infinite loops is unexpected defect, so a property to be held in the system is that any infinite loop never happens. When the property is satisfied, the model is excluded from targets being checked.

Procedure 3: When the property is not satisfied, the model is expected to include causes of defects. However, the obtained counter-examples may not cause the defects in the actual system. Such a counter-example is called a spurious counter-example. On the other hand, a counter-example that causes the defects in the actual system is called a real counter-example. So, we need to make it clear whether the obtained counter-example is spurious one or real one. When it is a real counter-example, causes of the defects are identified by analyzing the related state transition conditions.

Procedure 4: When it is a spurious counter-example, a strict model needs to be added to the initial model, so that counter-examples always cause the defects in the actual system. Returning to procedure 2, the same property is checked on the modified model. Until we can obtain real counter-examples, the strict model needs to be defined by translating the real source codes.

3.3 Model of “if” Statement and “for” Statement

Figure 2 shows a model of the “if” statement. The top location expresses the start state of the “if” statement. This conditional expression is also translated into two transitions representing the true state [a] and the false state [b]. The body of the “if” statement is translated into the transition [c], which expresses the end point of the “if” statement. The transition [d] is replaced by the “else” statement.

3.4 Abstraction of Other Statements

The other statements are translated into the UPPAAL model according to the following rules. Firstly, statements that do not include any variables in the conditional expressions of the “if” or “for” statements are excluded from the UPPAAL model. This is because these statements have no influence on the execution of the control flow model.

Secondly, statements that include variables in the conditional expressions of the “if” or “for” statements are translated into statements in the UPPAAL model. Function call expressions are temporally defined by variations in the returned values. Various kinds of collection data types that are not included in UPPAAL are temporally defined by an array in this model.

4 Defect Detection

4.1 Inspection Process

Inspection processes are divided into two parts, as shown in Figure 4. Firstly, for each method in the target source code, our supporting tool automatically generates several graphs. Each graph is a representation of a method that has several loop structures and conditions by abstracting the target
source code from the “if” statement and the “for” statement. Because the inspector can understand the basic specifications of the target source code, he/she can also understand the phenomenon caused by the defects. The defects being considered for inspection can then be selected from the structured diagram using the above methods. The UPPAAL model of the selected component is generated automatically. The Guard expression and the Update expression on each transition arrow are generated from the expression of the “if” statement and all expressions in the body.

When the generated expression is not suitable to the UPPAAL syntax rule, the inspector needs to modify it. Finally, the inspector defines a model for generating input data according to the expressions over a transition arrow. The inspector then starts to check the model using an expression that defines the occurrence of the infinite loop.

5.2 Initial Model Generation

Following the process described in Section 3 the support tool generates a structure diagram. This model is an initial model mentioned in Procedure 1 of Section 3.2. It is reasonable to assume that the inspector is familiar with the basic design of the target program, enabling a proper selection of the candidates for inspection. In this case, five candidates are found. Every candidate has a partial iteration, which may cause an infinite loop. An UPPAAL model to be inspected is automatically created from each candidate and in this case is called Model 1.

The aim of Model 1 is to confirm whether the infinite-loop state is reachable.

The inspector is able to determine the loop frequency owing to his/her understanding of the basic specifications. Therefore, in this model, we can assume that if the loop count is over 100, the transition of process 1 synchronizes with process 2. In this model, “e!” denotes message sending and “e?” denotes message receiving. The expression “I >= 100” is a guard condition of the transition. When conditions cause infinite loop, execution of the model will be stopped at the INFINITY_LOOP location. This model is named Model 2.

5.3 Confirmation of the Phenomena

Next, according to Procedure 2 mentioned in Section 3.2, the phenomenon is verified. Figure 5 shows how Model 1 and Model 2 are joined to yield a joined model. Figure 5 shows the integrated model for Case (1). The models for the other four cases are created in the same manner as is done for Case (1).

In order to confirm the occurrence of an infinite loop in each case, we define the path formula for checking the model in UPPAAL, as shown in Figure 6. In Case (1), the path formula shows the
INFINITY_LOOP location in process 2 is reachable through some path. Figure 6 shows this result. In the other cases (see Figure 7), the result shows that no path can lead to the INFINITY_LOOP location, and in these cases, the infinite loop is never observed. This warrants an investigation into the details of Case (1).

5.4 Judgment of a Counter-Example

In Case (1), there is a counter-example. According to Procedure 3 mentioned in Section 3.2, it needs making clear whether the obtained counter-example is spurious one or real one. As a result of analyzing a counter-example, it has become clear that there are some incorrect input data in the actual system. We could see that it is a spurious counter-example.

5.5 Generation of a Detailed Model

According to Procedure 4, we define input data model in the actual system, so that counter-examples always cause the defects in the actual system. To inspect the details, the method call expressions are replaced by the expressions of the method body. In the previous step, a method call expression is replaced by a variable.

In this step, the variable is restored to the original method call expression. As a result, these expressions with variables included in the conditional expressions of the “if” or “for” statements can be checked.

5.6 Creation of the Input Data Model

To do model checking the detailed model, a test model needs adding the model as the input data (see Figure 8). There are several constraints on the input data. Both the “From” period and the “To” period have values between 1 and 12. Furthermore, the “from” period is less than the “to” period. This model is named Model 3.

5.7 Model Checking

Figure 9 shows the model formed by joining Models 1, 2, and 3. Returning to procedure 2, the same property is checked on the modified model. An appropriate input value specifies the part where an infinite loop occurs. This path formula expresses the condition under which the INFINITY_LOOP location in process 2 is not reachable through all paths in Process 1. The result shows the existence of a traceable path to the INFINITY_LOOP location. This provides evidence for the occurrence of an infinite loop under certain conditions; the cause of this occurrence needs to be confirmed by examining the counter-example.
5.8 Model Modification and Rechecking

According to Procedure 3 mentioned in Section 3.2, it is confirmed that the obtained counter-example is real one. If we check the counter-example with the simulator screen of UPPAAL, It turns out that a large amount of search range is made, the search range is "From" period = 13 and "To" period=1. The cause has become clear from the counter-example, and the place of the defect of the source code could be specified. In this case, based on the above mentioned data, a path which creates it can be detected in the model. As the content of the record is “From” period=13, and “To” period=1, the path from id60 to id80 is the doubtful path. Thus, the cause of the infinite loop is discovered in the previous Guard conditional expression. That is, the expression on the path from id50 to id60 is a doubtful expression. "accPeriodTo==1" is unnecessary in conditional expression "(accPeriodFrom == 1 || accPeriodTo == 1)". We have to change the condition expression "(accPeriodFrom == 1)". The altered conditional expression is shown in Figure 10.

```
"(accPeriodFrom == 1 || accPeriodTo == 1)"
```

Finally, we can confirm that an infinite loop never occurs in the modified model. We define a test expression stating “The location INFINITY_LOOP is not true in all reachable states,” as shown in Figure 11. This path formula expresses that the INFINITY_LOOP location in process 2 is not reachable through all paths in process 1. The result “satisfied” shows that any path is not reachable to INFINITY_LOOP location, that is, any infinite loop never happens in this model.

```
Figure 10. Statement Change

"(accPeriodFrom == 1)"
```

```
Figure 11. Check Result

A not Process2.INFINITY_LOOP

Property is satisfied.
```

6 Discussion

Depending on the skill of the software developer, there are often many defects that avoid detection by code review or testing. Such defects cause various types of problems, as described below.

The case study discussed in Section 5 is a typical example of a failure caused by an infinite loop occurring under specific conditions. Because such a failure is caused by software development decisions that dishonor the designer’s original intention, we cannot guarantee the identification of a genuine case even if an inspector can intuitively detect the cause for the failure. Furthermore, such vexing causes of failure do not always reveal themselves in a consistent manner. Moreover, even if the correct cause is identified, it cannot be guaranteed that the same cause will produce a similar defect in the future. Our stepwise exhaustive checking method solves these problems and helps reduce the maintenance cost.

We have also applied our method to such embedded system developments as a line trace robot. In this case, time restriction in the UPPAAL model is useful in modeling the aforementioned robot. In embedded system development, it is often difficult to decide whether failure is caused by the hardware or by the software. Because the input and output values from sensors and actuators are sometimes different from the values assumed by the software, it is hard to reproduce the defect conditions. However, our method can be used to detect the cause by modeling the part of the system consisting of the robot and the running course.

7 Related Work

Achenbach [6] compared the abstraction techniques in various model-checking tools and applied these tools to real-world problems. This approach is very similar to ours. However, this paper only discusses an engineering method for the use of model-checking tools in the upper process of software development, omitting any method for detecting defects from the source code.

Bao [7] and Achenbach [6] suggested a procedure for abstracting the program to pay attention to the state that needs to be confirmed. This approach is very similar to ours. However, this paper only discusses an engineering method for the use of model-checking tools in the upper process of software development, omitting any method for detecting defects from the source code.

Thomas [8] has provided an UPPAAL model that has been correctly translated from the Java source code for real-time embedded systems. We have proposed several types of abstraction models for enterprise systems, but they have not proposed any guidelines for abstraction, for the sake of avoiding explosion of the number of states in the system.

Jianguo [9] and Bradbury [10] aimed to improve the precision of the verification process through model checking and model testing. However, they could not design an accurate model translated from the source code including the defects.

Salman [11] inspect the deadlock situation caused by Byte-range Lock between two or more processes by MPI (Message Passing Interface) using SPIN. He proposes two kinds of solutions, and compares the results by using model checking. The result shows...
that model checking is useful to make it clear which logic is effective. However, he did not show the way to investigate the difference between the performance characteristics.

Moonzoo [12] has proposed the framework for reflecting in an actual code the analysis result of model checking which used SPIN. He has shown the validity of a framework by introducing three case studies. As codes that have no need to be investigated are translated into a model, it may cause state space explosion. Although exclusion of an unnecessary portion is also possible by setting up a translation table, the great man day is required.

8 Conclusions

We have developed a technique based on model checking and selection of the necessary elements for identifying hard-to-detect defects in enterprise systems and for detecting the root cause of these defects. The results of a case study reveal that if the program is based on simple logic, only a small number of statements need to be abstracted from the source code. The inspector may face difficulties if the variables and input values in UPPAAL require to be defined.

In the future, we plan to develop a tool support and make improvements to the proposed method so that it can be used to treat more complicated models.

This proposal technique is applicable also not only in Java but other languages, if there is preparation of syntactic analysis. There is the method of syntactic analysis also about ABAP which is the development language of the case study. We are planning to improve the support tool so as to apply our method to ABAP programs.

References: