Design and implementation of fault injection based on abstract syntax tree of C Program

Hang Yu¹, Huiquan Gong² and Yawen Wang¹,*

¹State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, 100876, Beijing, China
²Faculty of Information Technology, Beijing University of Technology, 100022, Beijing, China

*Email: wangyawen@bupt.edu.cn

Abstract. Mutation testing is a fault-based technique to evaluate test quality by injecting bugs in programs under test. Traditional manual fault injection has many disadvantages such as inefficiency, large workload, low compilation success rate and so on. To improve the efficiency of fault injection, this paper aims to develop an automatic tool that supports fault injection for programs written in C. This paper first introduces the types of mutation operators and injection rules. Then, it proposes a framework, which mainly includes four modules: parser, fault location extraction, fault generation and data model. Finally, experiments show that our tool not only aids to produce a large number of program faults but also provides syntax-correct mutant programs, which would be further used in mutation testing.

1. Introduction

To improve safety and reliability of the system, it is necessary to pass software test to detect potential faults in the program before putting it into production. Mutation testing is a widely used test method. As a white-box unit testing technology, it has designed corresponding fault injection rules for a variety of programming languages. For example, Fortran[1], C[2] and Java[3]. The core idea of mutation testing is to inject manually generated faults into the program, and then test these injected faults using test suite. If the test can find all injected faults, then the test suite is sufficient.

Although the mutation testing can effectively evaluate the fault detection capability of the test suite, it also faces performance problems. One problem that prevents mutation testing from becoming a practical testing technique is that even for a small program, it is still possible to generate a large number of fault programs. At present, many people have been researching cost reduction techniques, it is divided into three types by Offutt and Unitch[4].

Another issue is related to labor costs such as human prejudge problems and equivalent mutants[5]. Human prejudge problem refers to verifying the output process of the source program in each case. The equivalent mutants mean that the two pieces to code have different syntax, but the semantics are the same. It is an undecidable problem and generally needs to be done manually, which is very time consuming and inefficient. Offutt found through empirical research that among a large number of variants generated, equivalent mutants generally account for 10% to 40%[6].

This paper implements an automatic fault injection tool to design a rapid and effective fault generation method by accelerating the fault injection method, the fault injection process and the
2. Design of automatic fault injection

2.1. Mutation operators and injection rules

In the mutation test, the process of fault injection is essentially the process of grammatical conversion of the program code. The grammar rule used to describe the conversion process is called the mutation operator. There are 18 mutation operators\[7\] used in this paper. They are AROR, ASOR, BIOR, CORP, LOOR, REOR, VARP, TIAS, TDBS, TDGS, TDCS, LOCN, AER, STER, RERP, LORP, BSRP and CSRP.

When injecting a fault, it is generally required to abide by the following rules: 1) The injected fault should be similar to the error that may occur when the programmer writes the code; 2) The injected fault must be grammatically and semantically correct, that is, the generated fault code can be compiled and passed; 3) The injected fault can be tested, that is, some algorithms or techniques can detect these faults, and the difficulty of detecting the injected faults cannot be too simple. Otherwise, it is easy to bring a certain difficulty to the accurate measurement of the positioning effects.

2.2. Business process design of fault injection

After analyzing the relevant mutation operator[7] and according to the structure of the abstract syntax tree, the automated fault injection framework can be divided into four modules.

1) Parser module: This module calls the C compiler and extracts information such as the abstract syntax tree and code text. There are already many mature tools that can do this.

2) Fault location extraction module: The input of the module is an abstract syntax tree of the source code and a mutation operator, and output is a set of syntax tree nodes and fault can be injected at the location of a node.

3) Fault generation module: The input of the module is the output of the previous module and the corresponding mutation operator. And the output is a fault set that can be injected at the position. A fault is represented by a text replacement instruction that represents a string operation, including insertion, deletion, and replacement.

4) Data model module: In the process of fault injection, we have designed many data structures and defined several operations on them. They each serve a specific module, and some run through several modules. This module will be described separately in the following subsections.

```
Figure 1. Fault injection process.
```

Figure 1 shows the specific business process. First, the parser module can get the abstract syntax tree of the program to be injected. Then, based on the given mutation operator, fault location extraction module analyzes the syntax tree structure and extracts the syntax tree node corresponding to the fault location. And it outputs a fault source object set indicating a fault injection position. And then, the fault generation module analyzes the mutation operator to be injected and the extracted fault location, generates a corresponding replacement code and outputs the variant object in the form of a text replacement instruction. Finally, the generated fault text replacement instruction is applied to the source file code portion to generate the corresponding fault program.
2.3. Design of fault location extraction

Fault injection needs to extract the abstract syntax tree first, and then extracts the fault location based on this. So the next step is to design a function to identify and extract the set of fault location nodes from the abstract syntax tree. For example, when the AROR mutation operator is specified, the abstract syntax tree will be traversed in breadth, and each SimpleNode will be judged to determine whether it conforms to the rules of AROR injection.

Figure 2 shows the specific process. In this process, range option requires users to specify the scope of the program to be injected when using the module interface, and the type option requires users to specify mutation operator to be injected into the fault. The last return option provides two types of fault location output mode, first is to output directly through the return value, and the other is that user provides the fault location manager object, and the module adds extracted fault location to the manager.

2.4. Design of fault generation

After extracting the fault injection location, the next step is to generate the fault code. The specific process is to calculate the corresponding fault replacement code according to the code conversion rules of mutation operator on the given fault source, and output the corresponding text conversion instruction. So the end result is a text conversion instruction. Essentially, the process of program mutation is the process of program text replacement.

The specific process is shown in figure 3. First, the fault output module receives the fault source manager as an input, traverses and extracts the fault source object in the manager space; and then, analyzes the source code text and the location information provided by the fault source. After that, generates the corresponding replacement code. Finally, outputs the text replacement instruction set corresponding to the fault source.

2.5. Data model

The automatic fault injection tool involves the following main data structures:

1) Abstract syntax tree node SimpleNode: it’s an abstract data type representing the structure of the program code.

2) Text index library TextIndexLib: this class provides a mapping from the code index of the SimpleNode to the string order of the program code. It ensures the correspondence between the syntax tree node and the program code.

3) File function mapper FileFunctionMapper: it provides the access interface for the fault injection process. For example, it can provide information such as code text, an abstract syntax tree and so on.
4) Fault type MutantType: the enumeration type, that is, the 18 mutation operators described above, used to describe the type of mutation operator.

5) Fault source MutantSourceItem: it indicates the location of the fault to be injected, and it includes two basic attributes, namely the mutation operator type and the fault location node.

6) Code change instruction TextCommand: this instruction regards the fault generation process as a string replacement operation on the original code, and it has three attributes: command, location and text. The command is a string operation instruction, which may be replacement, delete or insert. The location is the position of the string to be replaced and text is a replacement string.

3. Experimental test and result analysis

3.1. Case study

As shown in figure 4, this is a very simple code that defines three variables and has a nested loop. Then, we performed several tests on this code and analyzed the results.

As shown in table 1, it gives the mutation operator and the number and location of faults corresponding to it. Six mutation operators are specified for this code, namely VARP, AROR, TIAS, LOCN, REOR and CORP. The fault location is represented by a binary coordinate (line, column), where line represents the number of rows in which the fault is located, and column represents the number of columns of the starting character at the location of the fault.

| Mutation operator | Number of fault locations | Fault location |
|-------------------|---------------------------|----------------|
| VARP              | 7                         | (7,4) (5,10) (5,6) (6,11) (6,15) (6,14) (6,7) |
| AROR              | 4                         | (3,14) (5,14) (6,15) (7,14) |
| TIAS              | 2                         | (6,3) (5,2) |
| LOCN              | 2                         | (5,10) (6,11) |
| REOR              | 2                         | (5,10) (6,11) |
| CORP              | 7                         | (5,8) (6,13) (6,9) (3,16) (3,14) (5,12) (9,9) |

Table 2 shows partial mutation operators and its corresponding fault location and replacement instructions. The format of the replacement code is command (start, end, text), and the command is a text replacement command. There are three types of REPLACE, INSERT, and DELETE. start indicates the starting position of the fault to be injected into the source code, and end indicates the end position. And text is the fault code to be replaced.
Table 2. Partial test results.

| Mutation operator | Fault location | Replacement code |
|-------------------|----------------|------------------|
| AROP              | 5,14           | [REPLACE(55, 57, “--”)]| |
|                   | 6,15           | [REPLACE(74, 76, “--”)] | |
|                   | 7,4            | [REPLACE(86, 88, “--”)] | |
|                   | 3,14           | [REPLACE(36, 37, “+”), REPLACE(36, 37, “%”), REPLACE(36, 37, “/”)] | |
| TIAS              | 6,3            | [INSERT(81, “;”)] | |
|                   | 5,2            | [INSERT(61, “;”)] | |

At the same time, it generated 41 fault files. Comparing the results of the above two tables, we can find that the generated fault codes and their numbers are correct.

3.2. Project experiment

We tested three projects, qlib, deco and aa200c. Table 3 shows partial test results of aa200c.

Table 3. Partial test results of aa200c.

| File   | Func     | Killed | Survivor | Invalid | Valid | Score         |
|--------|----------|--------|----------|---------|-------|---------------|
| rotate | epmat    | 1,932  | 0        | 12      | 1,932 | 1             |
| moonjpl| domoon   | 7,632  | 0        | 49      | 7,632 | 1             |
| deltat | update   | 1,144  | 0        | 7       | 1,144 | 1             |
| rplanet| doplanet | 85     | 0        | 0       | 85    | 1             |
| nutate | nutate   | 3,431  | 1        | 0       | 3,432 | 0.999,708,6  |
| dms    | dms      | 859    | 89       | 91      | 948   | 0.906,118,1   |

File indicates the source file name in aa200c, Func is the function name in the file, Killed and Survivor respectively indicate the number of faults that have not been detected and detected. Invalid indicates the number of fault programs that have compiled errors, and Valid indicates the number of fault programs compiled successfully. Score indicates the fault detection rate.

As can be seen from Table 4, the tool generated a large number of fault programs, and the compilation success rate is also high. Except for a few generated fault programs that cannot be compiled, most of them are valid and can be used for mutation testing.

Table 4. Fault compilation accuracy rate of the tested project.

| Project name | Total number of generated faults | Compilable fault number | Invalid number of faults | Compilation success rate |
|--------------|---------------------------------|-------------------------|--------------------------|--------------------------|
| aa200c       | 127,386                         | 124,083                 | 3,303                    | 97.407%                  |
| deco         | 12,708                          | 12,538                  | 170                      | 98.662%                  |
| qlib         | 7,320                           | 7,000                   | 320                      | 95.628%                  |
According to the above two tables, we can see that the automatic fault injection tool realizes a predetermined function. It can generate a large number of fault programs, which have a high compilation success rate.

4. Conclusion
This paper implements an automatic fault injection tool. It has been proved by experiments that automatic fault injection greatly improves efficiency and has a high compilation success rate. However, this tool still has some shortcomings. It cannot eliminate equivalent faults, and it currently does not support statement substitution faults, because the 18 mutation operators[7] implemented by the tool only cover 59 of the 77 mutation operators of the standard C89. These things need to be further researched.

This paper is supported in part by the National Natural Science Foundation of China under Grant No.U1736110 and No.61702044.

References
[1] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, Mutation Analysis (Technical Report GIT-ICS-79/08, Georgia Inst. of Technology, 1979)
[2] H. Agrawal, R.A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, and E. Spafford, Design of Mutant Operators for the C Programming Language (Technical Report SERC-TR-41-P, Purdue Univ., 1989)
[3] P. Chevalley, Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach , Proc. Eighth Asia-Pacific Software Eng. Conf., 267 (2001)
[4] A. J. Offutt, R. H. Untch, Mutation 2000 : Uniting the Orthogonal, Proceedings of the 1st Workshop on Mutation Analysis, 34-44,(2001)
[5] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Trans. Software Eng., 37:649-78,(2011)
[6] Offutt J, Pan Jie. Automatically detecting equivalent mutants and infeasible paths[J]. Software Testing, Verification and Reliability, 165 (1997)
[7] H. Song, Y. Wang, Y. Gong, A Mutation-based Fault Injection Method for C Program, (2014). http://www.paper.edu.cn/releasepaper/content/201412-725