BoolMuTest: A Prototype Tool for Fault-Based Boolean-Specification Testing

Ziyuan Wang* Min Yu
School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China
*Corresponding: wangziyuan@njupt.edu.cn

***Abstract*—In order to perform mutation testing for general-form Boolean specifications, a prototype tool called *BoolMuTest* is designed for fault-based Boolean-specification testing. There are several function modules including generating mutants for general-form Boolean expressions, finding all possible test cases to kill a mutant, analyzing minimal failure-causing schemas for a mutant, calculating mutation score for a give set of mutants, and etc. All these functions are provided via command-line programs.**

***Index Terms*—Software testing, Boolean-specification testing, mutation testing, fault-based testing, prototype tool.**

I. Introduction

Fault-based Boolean-specification testing is one of important weak mutation testing techniques, since the running paths of program are usually dependent on Boolean-specifications in predicates. People usually pay their attention on 10 mutation types including ASF, CCF, CDF, ENF, LNF, LRF, MLF, ORF, SA0, and SA1 in the field of Boolean-specification testing [1]. In order to perform mutation testing for general-form Boolean specifications, a prototype tool called *BoolMuTest* is designed. It can be utilized to generate mutants for general-form Boolean expressions, find all possible test cases to kill a mutant, analyze minimal failure-causing schemas for a mutant, calculate mutation score for a give set of mutants, and etc.

II. Function Modules

A. *CreateBoolMutant*

CreateBoolMutant can be utilized to generate mutants for given general-form Boolean expressions with given mutation types. The command format is:

CreateBoolMutant origin_expr_file mutant_type_file mutant_expr_file [-disp]

Where the origin_expr_file is an input file including original Boolean expressions. The mutant_type_file is an input file including mutation types. The mutant_expr_file is output file including mutant expressions with given mutation types. And the [-disp] is an option parameter to print all the expressions (include original and mutant expressions) on the screen.

An example input file including original Boolean expressions is shown as follow, where there are two original Boolean expressions. To represent Boolean expressions by plain text, operators $\wedge$, $\vee$, and $\neg$ are replaced by *, +, and ! separately.

```
a*(!b+!c)*d+e
a1*!a5+(!a2+!a3+!a1)*a4+a5
```

An example input file including mutation types is shown as follow, where there are two types ASF and ENF.

```
ASF
ENF
```

The output file generated by *CreateBoolMutant* for above two input files should is shown as follow.

```
#Original Expression File: input.txt
#Original Expression 1: a*(!b+!c)*d+e
#Mutation Type: ASF
a*!b+!c*d+e
#End Mutation Type
#Mutation Type: ENF
a*!(!b+!c)*d+e
#End Mutation Type
#End Original Expression
#Original Expression 2: a1*!a5+(!a2+!a3+!a1)*a4+a5
...
```

B. *BoolCodeTransform*

BoolCodeTransform can be utilized to translate original Boolean expressions or mutant Boolean expressions in a given file to C language codes. The command format And is:

BoolCodeTransform expr_file code_file [-mu]

If [-mu] is used, mutant expressions will be translated; otherwise, original expressions will be translated. The expr_file is an input file that includes Boolean expressions, and the code_file is a C language header file.

By assuming the name of file that contain original expressions is "input.txt", the C code file for original Boolean expressions in section III.A should be:

```
bool input1(bool a, bool b, bool c, bool d, bool e)
{
return a&&(!b||!c)&&d||e;
}
bool input2(bool a1,bool a2,bool a3,bool a4,bool a5)
{
return a1&&!a5||(!a2||!a3||!a1)&&a4||a5;
}
```

And the code file for first two mutant Boolean expressions in section III.A should be:

```
bool input1ASF1(bool a, bool b, bool c, bool d, bool e)
{
return a&&!b||!c&&d||e;
}
bool input1ENF1(bool a, bool b, bool c, bool d, bool e)
{
return a&&!(!b||!c)&&d||e;
}
```

DOI reference number: 10.18293/SEKE2018-325

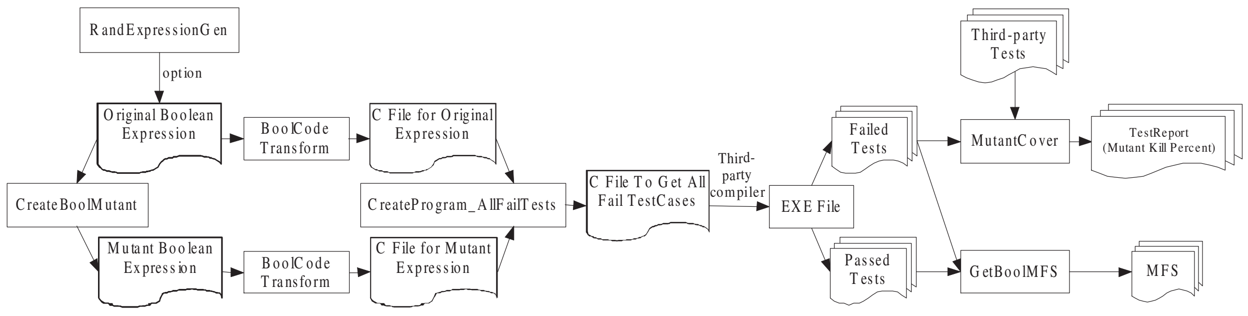
RandExpressionGen
option
Original Boolean Expression
BoolCode Transform
C File for Original Expression
CreateBoolMutant
CreateProgram_AllFailTests
C File To Get All Fail TestCases
Third-party compiler
EXE File
Mutant Boolean Expression
BoolCode Transform
C File for Mutant Expression
Third-party Tests
Failed Tests
Passed Tests
MutantCover
TestReport (Mutant Kill Percent)
GetBoolMFS
MFS

Figure.1 Work process of BoolMuTest

C. CreateProgram_AllFailTests

CreateProgram_AllFailTests can be utilized to create a C++ program that finds all possible test cases to kill given mutants. The command format is:

> *CreateProgram* origin_expr_code_file mutant_expr_code_file cpp_program_file [-W]|[-WO]|[-L]|[-LO]

Where both origin_expr_code_file and mutant_expr_code_file are input files generated by *BoolCodeTransform*. If [-W] or [-WO] is used, the created C++ program could be compiled and run in the Windows environment. If [-L] or [-LO] is used, it could be compiled and run in the Linux environment.

The C++ program, we named as *GetAllFailTests*, could be run as the following format on Windows/Linux console:

> *GetAllFailTests* [-i]|[-b] target_directory

Where the [-b] means that test cases are stored as binary string, and [-i] means that test cases are stored as an integer. For each mutant, all test cases that kill such a mutant are stored in an independent file in the folder arget_directory.

D. GetBoolMFS

The minimal failure-causing schemas reflect characteristic of failed test cases [2]. *GetBoolMFS* can be utilized to find minimal failure-causing schemas by comparing passed test cases and failed test cases. The command format is:

> *GetBoolMFS* expr_common_name min_expr_index - max_expr_index test_directory target_directory [{ASF | CCF | CDF | ENF | LNF | LRF | MLF | ORF | SA0 | SA1 | VNF | VRF} [min_mutant_index - max_mutant_index] | [min_mutant_index –] | [mutant_index]]*

Where expr_common_name is used to identify input files. Parameters min_expr_index and max_expr_index indicate the range of index of original expressions. Parameters test_directory and target_directory indicate folders that store failed test cases and final results. And parameters min_mutant_index and max_mutant_index indicate range of index of mutant expressions.

E. MutantCover

In mutation testing, the mutation score is used to evaluate quality of given test cases. *MutantCover* can be utilized to evaluate percent of killed mutants in Boolean-specification testing. The command format is:

> *MutantCover* origin_expr_name mutant_type all_fail_tests_dir input_tests_file

Where parameter origin_expr_name and mutant_type are used to indicate mutations. The folder all_fail_tests_dir store files that contain all failed test cases for each mutant. Test cases under evaluation are stored in input_tests_file.

F. RandExpressionGen

There is a program called *RandExpressionGen* to generate Boolean expressions randomly according to given configurations. The usage of *RandExpressionGen* is omitted here since the limitation of the length of pages.

III. Conclusion

The work process of *BoolMuTest*, which is a prototype tool for fault-based Boolean-specification testing, could be found in Figure 1. The *BoolMuTest*, which provides six command-line programs, supported many previous experimental studies in the field of fault-based Boolean-specification testing [3][4][5]. There should be some future works of making the tool user friendly by design graphic user interface. The current version could be found in https://github.com/princeyuan/BoolTest/.

Acknowledgment

This work is supported by the National Nature Science Foundation of China (61772259).

References

[1] Z. Chen, T. Y. Chen, B. Xu. A Revisit of Fault Class Hierarchies in General Boolean Specifications. ACM Transactions on Software Engineering and Methodology (TOSEM), 2011, 20(3): 13.

[2] C. Nie, H. Leung. The Minimal Failure-causing Schema of Combinatorial Testing. ACM Transactions on Software Engineering and Methodology (TOSEM), 2011, 20(4): 15.

[3] Ziyuan Wang, Yuanchao Qi. Why Combinatorial Testing Works: Analyzing Minimal Failure-Causing Schemas in Logic Expressions. 2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW2015): 4th International Workshop on Combinatorial Testing (IWCT2015).

[4] Chunrong Fang, Zhenyu Chen, Baowen Xu. Comparing Logic Coverage Criteria on Test Case Prioritization. Science China Information Science, 2012, 55(12): 2826-2840.

[5] Min Yu, Ziyuan Wang, Yuanchao Qi, Feiyan She, Weifeng Zhang. A Revisit of Fault-Detecting Probability of Combinatorial Testing for Boolean-Specifications. 30th International Conference on Software Engineering and Knowledge Engineering (SEKE2018).