ATLANTIS PRESS

Advances in Engineering Research, volume 138

2nd International Conference on Machinery, Electronics and Control Simulation (MECS 2017)

Compiler Testing Approach based on Generalized Equivalent Substitution

Xinwei Wu[a], Peng Zhang[b], Qiang Huang[c], Qirong Ma[d], Guanglei Shao[e], Pengcheng Li[f]

School of Jilin University, Changchun130012, China

[a]15526878862@163.com, [b]zhangpengccst@jlu.edu.cn, [c]15754311016@163.com, [d]18166891775@163.com, [e]894130459@qq.com, [f] 15754310995@163.com

Keywords: compiler, generalized equivalence substitution, mutation operator.

Abstract: The importance of the compiler is unquestionable. In order to better guarantee the correctness of the compiler, this paper presents a compiler testing approach based on generalized equivalent substitution. This approach tests the compiler by building a large number of equivalent assemblies. It can produce more test cases with fewer original test cases and effectively avoid a test oracle. We use the way implanting variants in SNL compilers to verify the validity of the compiler test approach based on generalized equivalence substitution. The experiment results show that our approach is effective and can detect errors in the compiler.

1. Introduction

The compiler plays an important role in software development. If the compiler error can lead to inconsistencies in the source code semantics and executable file behavior, this question is secret and deadly for programmer. In fact, the compiler does have an error [1]. Therefore, testing the correctness of the compiler is critical.

The traditional compiler testing technology tests the under test program by comparing the actual output of the test cases with the expected output [3], For the compiler testing, in many cases there is a test oracle that it is difficult for the tester to construct the expected output of the program to determine whether the execution result is correct [1,2,7]. At present, the three widely used compiler testing techniques include Randomized Differential Testing (RDT) [8], a variant of RDT—Different Optimization Levels (DOL) [9], and Equivalence Modulo Inputs (EMI) [6]. The three compiler testing techniques have their advantages and disadvantages, in which RDT is the most effective technique for error detection[6], but it relies on high-precision compilers[3], and the impact of oracle in EMI technology is minimal , But it is less efficient[6]. The Metamorphic Testing Technique [10] is to test the program by checking the relationship between multiple execution results of the program, without constructing the expected output [4], effectively avoiding the risk of a test oracle in software testing.

In this paper, the technique of metamorphic testing is applied to the compiler testing, and a new compiler test method based on generalized equivalence substitution is proposed. This approach uses a generalized equivalent substitution method to generate a large number of equivalent assemblies as test cases, so as to test the compiler under test by comparing the output elements within the relationship is equal. This approach can use a small number of original test cases to generate a large number of available test cases, but also can avoid the risk of a test oracle. In the generalized equivalence substitution, we propose several kinds of mutation operators, that is, the method of transforming the program. The mutation operator designed in this paper will get the equivalent program group after the transformation of the program. To demonstrate the effectiveness of this approach, we used the method of implanting variants to test the obtained SNL compiler mutant, which detected 30 errors in 32 mutants. The experimental results show that the compiler testing approach based on generalized equivalence substitution is effective.

Copyright © 2017, the Authors. Published by Atlantis Press.
This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

2. Basic Knowledge

Definition 1: Set up a program P written in the advanced programming language. E is the object code compiled by the compiler. The compilation process of the compiler can be regarded as a function: F: P → E.

Definition 2: If I is the input of program P and O is the corresponding output, then P (I) = O means that the output of P is O when input is I.

Definition 3: If there is mutation operator M and program P, then M (P) is the effect of M on P, where M (P) represents the new program obtained by the action of P.

Definition 4: When I is the input of program P, we call the set of executed statements in the program run as W (I, P), otherwise the rest is called non-execution path ¬ W (I, P).

Definition 5: If there is a mutation operator set $\{M_1, M_2, \ldots M_i\}$, for a program P of the certain input I, if $(M_1(P))\ (I) \equiv (M_2(P))\ (I) \equiv (M_3(P))\ (I) \equiv \cdots \equiv (M_i(P))\ (I)$, it is said that the effect of this mutation operator group on the program P with input I is equivalent.

Definition 6: Suppose that a statement S_2 in a program will be affected by statement S_1, then say S_2 depends on S_1.

3. Compiler Testing Framework Based on Generalized Equivalent Substitution

The basic principles of the testing framework are as follow: If there are original test cases P_1 and P_2, P_1 is equivalent with P_2 under the condition that the input is I , the output are equivalent after compiling $(F(P_1))(I) \equiv (F(P_2))(I)$. Because of the transitivity of equivalence relations, we can conclude that:

$$(F(P_1))(I) \equiv (F(P_2))(I) \equiv \cdots \equiv (F(P_n))(I) \quad (n>2)$$

Therefore, when we put the equivalent program group that contains the same input into the compiler for testing, we only need to compare the equivalent relationship between the outputs, it can avoid the risk of the Oracle problem in the test. The framework is illustrated in Figure 1 in this paper.

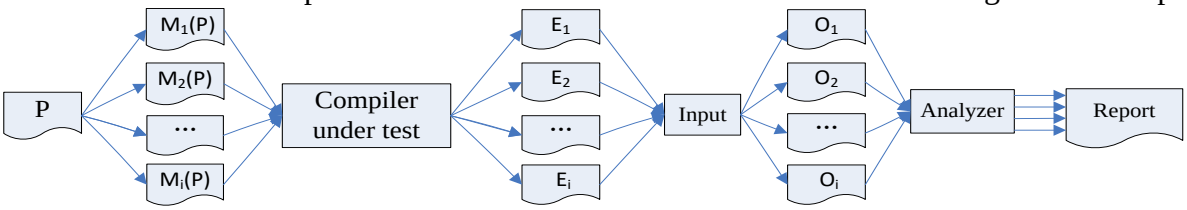

Figure.1 Compiler Testing Framework

For the original test case P, use the mutation operator method to manipulate P in the operator library , after the transformation of the equivalent variation operator$(M_1,M_2,\ldots M_i)$,it will produce a corresponding equivalent program group{ $M_1(P),M_2(P)$,…,$M_i(P)$}; Next, use the Compiler for testing to compile the equivalent program group to obtain the executable file group{$E_1,E_2,\ldots, E_i$};At the moment we regard I as the input of $E_1,E_2,\ldots E_i$, and get the set of output data {$O_1,O_2,\ldots O_i$}; At the same time judge the relationship between the various elements in the set, if the elements are all equal that it can prove that we did not detect the error in this experiment, otherwise the compiler exists error.

In the whole testing framework, the most difficult part is the generation of test cases. We put forward the idea of generalized equivalence substitution, designing three kinds of mutation operator method to manipulate the source program. We will describe our approach of the generation of test cases in detail in an upcoming section.

4. Generation of Test Cases

4.1 Original test cases

The method of generalized equivalence substitution does not need to guarantee the equivalence relation between the program which is after transformation and the source program. In terms of the characteristics of the method of generalized equivalence substitution, we do not propose special

requirements for the original test case, and we can use the programs generated by the program automatic generation tool such as C-smith.

4.2 Mutation Operator

Mutation operator refers to make equivalent transformation for a program. In order to satisfy the generalized equivalence substitution, we designed three kinds of mutation operators:

4.2.1 Augmentation Operator (AO)

That is, adding a single or compound unrelated statement in position in the source program. The method of generalized equivalence substitution does not need to guarantee that the semantics of the generated program are consistent with semantics of the source program, so we can directly manipulate the assignment or operation statement of the variable selected. Here we introduce a method of generating an irrelevant statement based on an equivalent expression: First, we need a set of expressions for a common identity equation, like:

{ $e \times 2 \equiv e \times e$,
$e_1 + e_2 \equiv e_2 + e_1$,
$(e_1 + e_2) \times (e_1 + e_2) \equiv e_1 \times e_1 + 2 \times e_1 \times e_2 + e_2 \times e_2$,
…… }

Figure 2 shows an example of the generation of an equivalent expression set, and we will put the irrelevant variable x, y needed to be generated into the set :put x into $e \times 2 \equiv e \times e$, get $x \times 2 \equiv x \times x$ (1)；then put（1）and y into $e_1 + e_2 \equiv e_2 + e_1$,get $x \times 2 + y \equiv y + x \times x$ (2),by the iteration method, we can get an equivalent expression set.

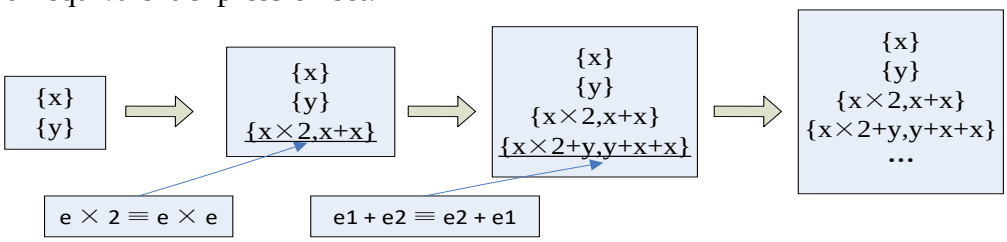

Figure.2 build an equivalent expression set

Set an unrelated variable V, generate a V equivalent expression set S (V) in the above way, and fill the elements of S (V) randomly into the program to obtain a large number of equivalent programs.

4.2.2 Elimination Operator (EO)

As the words suggest, it means that eliminate the irrelevant statement. In the case of determining the output, some statements in the program are not executed. The source program P is put into compiler C to be compiled to obtain the executable program O, the process can be represented by the function F (P) → O. Put the input I into the executable file O, in the process of running the executable program, we can locate the statements in the source program that can be used, the set of these statements is called Execution path under input I S(I,P), On the contrary, the rest is called a non-execution path ¬ S (I, P), and the source program P can be expressed as P = S (I, P) ∪ ¬ S (I, P).

It should be noted that the operation of the deletion should not affect the correctness of the program.

4.2.3 Modification Operator (MO)

We should change the statement without changing the semantics of the program. And we designed three modification operators.

①Replace the comput expressions equivalently. An expression contain an operator, marking the statement in the program's position. Converting the elements of the expression set to the marked statement to generate a new equivalent assembly.

② loop invariant hoist. There is a situation, in the process of cycling, a variable does not change the cycle, and then the variable on the assignment of the statement can be extracted outside the loop.

③ change the non-dependent statements. This paper focuses only on the existence of statement dependencies, without the need to distinguish the kind of dependencies. The statements does not change the semantics if there is no dependency between statements.

5. Experiment

To verify the effectiveness of our test approach, we made an experiment by implanting variants in the compiler. We chose the C ++ compiler and used to teach the SNL compiler as an experimental target. The compiler's source code comes from https://github.com/hikean/SNL-Compiler.The experiment is as follows:

We use the method of implanting variants to modify the source code of the SNL compiler, and set each mutant to produce a modified one. Specific modifications are divided into two categories: the first is the exchange of symbols, including: "+", "-", ">", "<"; the second category is to modify the value in the assignment statement. We screened all mutants that lost the essential function, and finally obtained 32 available mutants, we numbered these mutants {C_1, C_2,…, C_{32}}.

We use the compiler developer's seven SNL program examples as the original test case. We apply the mutation operator in the operator library to manipulate the original test case, Get 171 sets of equivalent assemblies. These assemblies are numbered, with the addition operator getting 110 groups {P_1, P_2,…, P_{110}}. We tested 32 mutants using this 171 group equivalence program. The final detection of 30 mutants in the BUG, the wrong detection rate of 93.75%. This indicates that the approach in this paper is valid and can detect bugs in the compiler. C19 and C26 in the two mutants of the Bug has not been detected, we think this is the number of experimental test cases caused by insufficient enough. We illustrate our experiment with two examples:

The mutant C_{14} selects the "genNum" function within the source code "tm.cpp" to modify, replacing all the '+' in the function with '-' (see Figure 3). Equivalent assembly of two equivalents in P_{142} (Figure 4), After compiling and inputting, the set of outputs is {..., -212, 4, ...}, and there is an unequal element in the set, which proves that the mutant has error.

```
do
{ sign = 1;
  while ( nonBlank() && ((ch == '+') || (ch == '-')) )
  { temp = FALSE ;
       if (ch == '-')  sign = - sign ;
    getCh();
  }
  term = 0 ;
  nonBlank();
  while (isdigit(ch))
  { temp = TRUE ;
    term = term * 10 + ( ch - '0' ) ;
    getCh();
  }
  num = num + (term * sign) ;
} while ( (nonBlank()) && ((ch == '+') || (ch == '-')) ) ;
```

Figure.3 tm.cpp

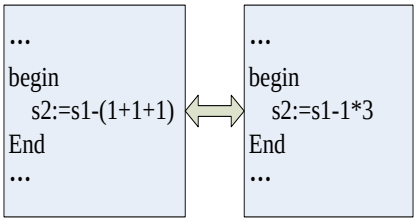

Figure.4 test cases in P_{142}

Mutate C_{31} selects the "genStmt" function in the source code "cgen.cpp" to modify the "p0 = t-> child [0];" statement in "case WhileK" to "p0 = t-> child [1]; "(Figure 5). Equivalent assembly P170 within the two procedures (Figure 6), after compilation and input, the output set contains a program grammar error, to prove that the mutant has error.

```
case WhileK:
        if ( TraceCode )   emitComment("->while");
        p0 = t->child[0];
        p1 = t->child[1];
        currentLoc = emitSkip(0);
        cGen(p0);
        savedLoc1 = emitSkip(1);
        emitComment(" while : jump out while ");
```

Figure.5 cgen.cpp

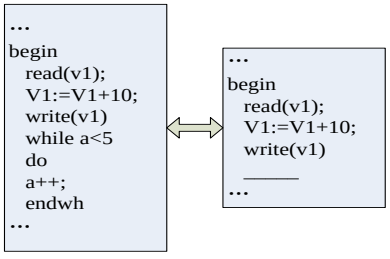

Figure. 6 test cases in P_{170}

Although the test approach based on generalized equivalence substitution can avoid the risk of a test oracle, but this contrast approach is random, there may be cases that the test case does not trigger the modified code in the mutant while compiling. In theory, a comprehensive type of original test cases and a sufficient number of test cases can solve this problem, but the increase in the number of cases will increase the workload and reduce efficiency, which is the limitations of our test methods.

At the same time in the experiment we also found that there are some problems: the same equivalent program group within the program, there is the situation where the output is inconsistent. After verification, we found the problem is that floating-point operations can not maintain the original value, so the equivalent of the results of the program is not the same situation. To avoid this problem, we only use integer variables and constants to operate.

6. Conclusion

The generalized equivalence substitution is obvious, and the generalized equivalence substitution does not need to qualify the semantics of the generator and the semantics of the source program. Therefore, only a few typical structure of the source program and a sufficient expression of the equality equation, you can get a lot of test cases. At the same time, we have experimentally proved that our compiler test approach based on generalized equivalence substitution can detect the errors in the compiler.

References

[1] Kossatchev A S, Posypkin M A. Survey of compiler testing methods [J]. Programming & Computer Software, 2005, 31(1): 10-19.
[2] Barr E T, Harman M, Mcminn P, et al. The Oracle Problem in Software Testing: A Survey [J]. IEEE Transactions on Software Engineering, 2015, 41(5): 507-525.
[3] Tao Q, Wu W, Zhao C, et al. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique[C]// Asia Pacific Software Engineering Conference. IEEE, 2010: 270-279.
[4] Yoshikawa T, Shimura K, Ozawa T. Random Program Generator for Java JIT Compiler Test System [C]// International Conference on Quality Software. IEEE Computer Society, 2003:20.
[5] Bird D L, Munoz C U. Automatic generation of random self-checking test cases [J]. Ibm Systems Journal, 1983, 22(3): 229-245.
[6] Chen J, Hu W, Hao D, et al. An empirical comparison of compiler testing techniques [C]// International Conference on Software Engineering. ACM, 2016: 180-190.
[7] Chen T Y, Cheung S C, Yiu S M. Metamorphic testing: a new approach for generating next test cases [J]. 1998.
[8] Sheridan F. Practical testing of a C99 compiler using output comparison [J]. Software Practice & Experience, 2007, 37(14): 1475–1488.
[9] Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers.[J]. Acm Sigplan Notices, 2011, 46(6): 283-294.
[10] Chen T Y, Kuo F C, Tse T H, et al. Metamorphic testing and beyond [C]// Eleventh International Workshop on Software Technology and Engineering Practice. IEEE Computer Society, 2004: 94-100.