Extending Equivalence Transformation Based Program Generator for Random Testing of C Compilers

Shogo Takakura
Kwansei Gakuin University
Sanda, Hyogo, Japan
s-takakura@kwansei.ac.jp

Mitsuyoshi Iwatsuji*
Kwansei Gakuin University
Sanda, Hyogo, Japan

Nagisa Ishiura
Kwansei Gakuin University
Sanda, Hyogo, Japan
nagisa.ishiura@ml.kwansei.ac.jp

ABSTRACT

This paper proposes a method of reinforcing random program generation for automated testing of C compilers. Although program generation based on equivalence transformation is a promising method for detecting deep bugs in compilers, the range of syntax it can cover has been narrower than the production rule based methods. While the conventional method based on equivalence transformation can only generate programs with scalar variables, assign statements, *if* and *for* statements, the proposed method attempts to extend them to handle arrays, structures, unions, as well as *while* and *switch* statements and function calls. A random test system, Orange4, extended with the proposed method has detected bugs in the latest development versions of GCC-8.0.0 and LLVM/Clang-6.0 which had been missed by the existing test methods.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Software performance*; *Software testing and debugging*;

KEYWORDS

compiler validation, automated random test, fuzzing, equivalence transformation, minimization

ACM Reference Format:
Shogo Takakura, Mitsuyoshi Iwatsuji, and Nagisa Ishiura. 2018. Extending Equivalence Transformation Based Program Generator for Random Testing of C Compilers. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '18), November 5, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3278186.3278188

1 INTRODUCTION

Since compilers are fundamental tools for software development, they must be highly reliable. While front-end modules for lexical and syntax analysis of the mature compilers are stable, middle-end and back-end modules in charge of optimization and code generation are continuously modified to incorporate aggressive and sophisticated optimization algorithms. This leads to frequent occurrence of bugs, and many bugs are actually being reported to their bug databases [1] [2].

The test of compilers usually relies on test suites, huge sets of test programs, such as [1–3, 5]. Although they are the collection of good test cases and compilers are tested efficiently, it is theoretically impossible to validate compilers completely with a finite number of test programs.

Automated random test is a powerful tool to compensate this weakness. Various random test methods of C compiler have been proposed so far, among which one of the outstanding ones is Csmith [14]. It is a comprehensive random test generator which covers the broad range of C language grammar. It detected 79 and 202 new bugs in GCC and LLVM during the three years to 2010 and greatly contributed to the improvement of the quality of these open source compilers.

Csmith basically generates random test programs based only on grammar rules, which is one of the reasons why it can cover wide range of syntax relatively easily. However, since it does not keep track of the execution result of the generated programs, they might end up with undefined behavior such as zero division or out of bounds access of arrays. To avoid generating invalid test programs with undefined behavior, Csmith imposes restrictions on the form of sentences and expressions, such as that division must always appear in subexpression like `(`B`!=0?`A/B`:`A`)`. Thus, while Csmith can cover a wide range of syntax, this kind of restriction limits the class of programs it can generate.

On the other hand, in Orange3 [10] and Orange4 [11], test programs are generated by using semantic information of the program in combination with the grammar rules. By keeping track of the values of all the variables and expressions in the program at program generation time, they avoid generation of illegal programs without adding restrictions on sentences and expressions. Especially, it can generate long and complex expressions for testing compiler arithmetic optimization, and they detected bugs in GCC and LLVM that had not been detected by Csmith.

In this approach, however, a data structure to represent the semantics of the generated program must be constructed along with the syntax tree, which complicates the program generation process. For this reason, Orange3 and Orange4 can generate programs in which variables are only scalar and control statements are only *if* and *for* statements.

*Currently with NTT PC Communications Inc., Tokyo, Japan.

A-TEST '18, November 5, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6053-1/18/11…$15.00
https://doi.org/10.1145/3278186.3278188

[1] https://gcc.gnu.org/bugzilla/
[2] https://bugs.llvm.org/

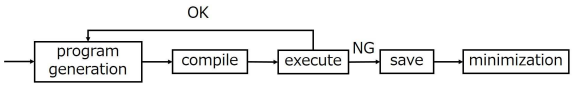

Figure 1: Random testing of compilers

```
 1: #pragma pack(push)
 2: #pragma pack(1)
 3: struct S1 {
 4:   volatile int32_t f0;
 5:   struct S0 f4;
 6:   volatile uint8_t f7;
 7: };
 8: #pragma pack(pop)
 9: union U3 {
10:   const uint32_t f0;
11:   const volatile int64_t f1;
12:   int32_t f2;
13: };
14: ...
15: static uint32_t * func_1(void)
16:
17: {
18:   ...
19:   if (func_2(g_31[3])) {
20:     for (i = 0; i < 1; i++)
21:       l_96[i] = &g_97;
22:     (****g_1658) = ((((safe_mod_func_int32_t_s_s(
23:     g_450[(g_133.f6 + 2)][(g_133.f6 + 3)], g_62)) >=
24:     l_172), 0x5318L), (g_1725, l_1726));
25:     (*p_61) = ((*g_821) |= (safe_lshift_func_uint8_t_u_u(
26:     ((safe_mul_func_uint8_t_u_u(g_22.f1, l_1371)) |
27:     ((++l_1664) > l_1667)), l_1336)));
28:   }
29:   ...
30: }
31: ...
32: int main ( int argc, char* argv[])
33: {
34:   ...
35:   func_1();
36:   ...
37: }
```

Figure 2: Test program generated by Csmith

In this paper, we propose a method of extending random C program generators that use semantic information with more sophisticated data types and control statements. As for the data types, arrays, structures, unions, and arbitrary nests of them can be generated. Regarding the control statements, function calls, *while* statements, and *switch* statements may be generated in addition to *if* and *for* statements.

We have implemented a random test system of C compilers on top of Orange4, which covers C syntax almost comparable to Csmith, and can generate a certain class of programs which can not be generated by Csmith. It successfully found error producing programs for the latest development versions of GCC-8.0.0 and LLVM/Clang-6.0, which are difficult to generate by the existing other methods.

2 RELATED WORK

2.1 Random Testing of Compilers

Figure 1 shows a general flow of compiler random testing. In the main loop, random program generation, compilation, and execution is repeated as long as time allows. If an error such as a crash of the compiler or an incorrect execution result is detected, the program is saved for later analysis. The error inducing test program is called an *error program*. The size of the random programs is adjusted to the compiler under test, but typically ranges from several hundred lines to several thousand lines. Since it is impossible to debug the compiler given such a large error program, it is *minimized* or reduced; a test program as small as possible and yet produces the error is searched.

There are two major challenges in compiler random testing: (1) how to decide if the execution results of randomly generated programs are correct, and (2) how to avoid generating undefined behavior such as zero division, integer overflow, out-of-bounds array access, etc.

There are two major approaches to deal with the first one; a differential testing method and an oracle based method. In the differential testing, the same test program is compiled with different compilers and the execution results (the outputs from the program) are compared. Csmith [14] is based on the differential testing method. On the other hand, in the oracle based method, the expected outputs are somehow known before or during program generation, based on which the correctness of the execution result is judged. Quest [9] for testing function calls and Orange3 [10] and Orange4 [11] for testing the arithmetic optimization fall in this category. In the differential testing, the second challenge of avoiding undefined behavior must be somehow resolved. One way is adding some restrictions on grammar rules. In oracle based method, it is relatively easy to avoid undefined behavior, but how to maintain the semantic information during program generation becomes an issue.

Regarding program generation, there are two approaches; one is based on production rules and the other is based on equivalent transformations on test programs. Quest, Csmith, Orange3 are based on the former approach. Proteus [8], Athena [7], and Hermes [13] generate new test cases from existing test programs (such as those generated by Csmith). Orange4 generates long random programs from an obvious seed program by repeated applications of equivalence transformations.

2.2 Csmith

An example of a test program generated by Csmith [14] is shown in Figure 2. Arrays, structures/unions, pointers as well as scalar variables are generated. Regarding control flow, *if* statements, *for* statements, function calls, *goto* statements, etc. are generated.

Since Csmith does not know the behavior of the test program at generation time, it is devised so that the program does not cause undefined behavior. For example, in line 26, `safe_mul_func_uint8_t_u_u` is a macro to perform multiplication, but it just returns the first operand if overflow is detected by examining the operands. So the operation is always guarded. To avoid an infinite loop, the loop bounds of *for* statements are constants, as in line 20. An array subscript is limited to a loop variable whose range is known (as in line 21), or a sum of a constant and a variable whose values is known. *While* and *switch* statements are not generated in Csmith, probably to avoid infinite *while* loops and *switch* statements which mostly select default clauses.

As a tool to minimize error programs detected by Csmith, C-Reduce [12] is used. It is a general-purpose minimization tool, which takes a source code of the C program and the command line for error judgment and outputs the reduced program that still fails the test. It tries to reduce the program by repeatedly applying size reducing transformations. The transformations may cause undefined behavior, which is checked by a static analyzer. If undefined behavior is detected, the application of the transformation is canceled

and another transformation is tried. Thus, minimization may take long time especially when undefined behavior occurs frequently.

2.3 Orange4

An example of a test program generated by Orange4 [11] is shown in Figure 3. Orange4 keeps track of the values of all variables and expressions at program generation time, so it can generate long and complex arithmetic expressions without guards to avoid undefined behavior. It can also generate expressions for the loop bounds (as in line 12). The test program judges whether the execution results are correct or not by itself comparing the values of the variables with the expected value as in lines 22–24. However, Orange4 can generate only *for* and *if* statements, and scalar variables.

In Orange4, large random test programs are generated by repeatedly applying equivalence transformation (statement addition and expression expansion) of a program from an obvious program that includes only "`return 0;`" in the main function. All the complex expressions are generated by equivalence transformations from constants (whose values are known, of course), as shown in Figure 4. The value (5) of the expression is first determined, and then it is expanded to an expression with the same value (2 + 3). This is repeated recursively. Undefined behavior is easily excluded by avoiding problematic operands. It is also possible to choose the boundary values as operands, which contributes to enhance bug detection capability.

As an internal representation during program generation (and also error program minimization), Orange4 builds an AST (abstract syntax tree) with the runtime value annotated to every variable and subexpression. The runtime value is unique, which means that Orange4 generates test programs in which each variable may be assigned plural times but the same value is assigned every time.

Orange4 is equipped with a built-in error program minimizer. Transformations to reduce program size are applied as long as the error remains. Note that the transformations are applied on the AST with semantic information. Since transformations that incurs undefined behavior are detected before application, backtracks regarding undefined behavior never happen.

While the semantic information attached to AST is beneficial in many ways, it is a burden when we want to handle broader syntax, for we must keep the runtime value of every variable unique.

3 EXTENSION OF DATA TYPES AND CONTROL STATEMENTS

In this paper, we propose a method of increasing the variety of data types and control statements in equivalence transformation based random program generation. As for data types, arrays, structures, and unions, and arbitrary nests of them are newly introduced. As new control statements, function calls, *while* statements, and *switch* statements are added.

3.1 Extension of Data Types

3.1.1 Outline. Even if aggregate and compound data types are introduced, it is a scalar member or a scalar element that is referenced in expressions. Thus, slight extension of the expression generation method of Orange4 is enough to handle those new data

```
 1: #define OK()
 2: #define NG(fmt,val) __builtin_abort()
 3: const volatile signed int x9 = -59;
 4: int main (void)
 5: {
 6:   static unsigned long long x0 = 7LLU;
 7:   static const volatile signed long x1 = 0L;
 8:   ...
 9:   int t0 = 46;
10:   signed long t1 = 297271L;
11:   ...
12:   for( i = x9*x6-x8; i < x5+x5; i -= x7+x3 ) {
13:     t0 = x3|x0*x2-x4;
14:     t1 = x5*x5-x6+x2/x7;
15:     if( x1<<x1 ) {
16:       t2 = x8>>x4/t0+x10*x10+x11;
17:     }
18:     else {
19:       t3 = x14|x16;
20:     }
21:   }
22:   if (t0 == 120) { OK(); } else { NG("%d", t6); }
23:   if (t1 == 220) { OK(); } else { NG("%d", t6); }
24:   if (t3 == 22) { OK(); } else { NG("%d", t6); }
25:   return 0;
26: }
```

Figure 3: Test program generated by Orange4

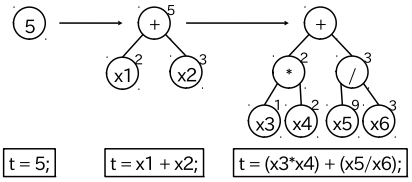

Figure 4: Derivation of expressions in Orange4

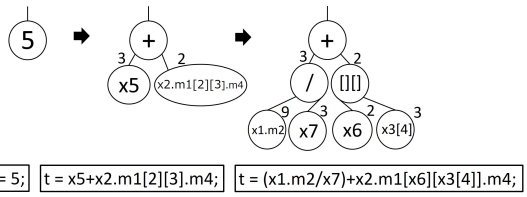

Figure 5: Extended version of expression derivation

types. Figure 5 shows an extended version of expression generation. When expanding constant 5 using addition, a member of nested data structure (`x2.m1[2][3].m4`) can be used in place of a scalar variable (in parallel with this, a data structure to represent the array and structures is build). In the next step, reference to the array element is regarded as a kind of arithmetic operation (`[][]`) and constants (2 and 3) are further expanded. Arrays, structures, and unions may recursively appear in these expressions.

In addition, aggregate variables may be passed as arguments to functions calls which will be discussed in the next subsection.

3.1.2 Arrays. An example of a program containing arrays generated by our method is shown in Figure 6. Arrays' dimensions, sizes, basic types, and initial values are randomly determined (as in lines 8–9). They may be either local or global (or static global). Access to array elements may appear in the same places in the program as scalar variables; it can be assigned and may be referenced multiple times (as in lines 11–14). Collation with expected values is also done to array elements (as in line 16).

```
 1: int func0(int a0[5], long a1) {
 2:   ...
 3:   t50 = a0[2] + x43;
 4:   if (t50 == -59000) {OK();} else {NG();}
 5:   return t50 + x51 - a0[4];
 6: }
 7: int main(void) {
 8:   int x6[5] = {1227,4,-59433,-106,24};
 9:   int t42[3][5] = { {-3290,559,23085,26668,-38567},
       {358148,8,4,35388,85600},
       {12594474,-293697,979,-125774,1138055} };
10:   ...
11:   t42[1][2] = (x4 - x3) - func0(x6, x11 * x12) ;// = 100
12:   unsigned int t48[x6[1] + t42[1][2]];
13:   t48[0] = x6[1] * x11;
14:   t42[x12][t4[4]] = (x5 && t48[x16 + t1]) + t2;
15:   ...
16:   if (t42[1][2] == 100) {OK();} else {NG();}
17: }
```

Figure 6: Test program with arrays

Subscripts may be general expressions and may contain again references to array elements (as well as to structures, unions). Note that they never cause out-of-bounds accesses, for they are expanded from constants within the array bounds. This is one of the powerful features of the proposed method which has not be realized in the other existing methods.

This feature is used also to generate variable-length arrays, as shown in line 12. An array may be declared with its size specified by an expression. This is possible because the run time value of the expression is controllable at generation time and never be negative nor too large.

Besides the element accesses, an entire array may be passed as an argument of functions (as in line 1). The array variable can be referenced in the function in the same way as local array variables.

How to realize generation of array variables is rather straightforward. Basically, each array elements is given an entry in variable instance table and the same information as a scalar variable (the lexical name, the initial value, the runtime value, etc.) is also defined to the array element. Passing of array arguments is also realized in a simple way. When generating a call to a function with aggregate arguments, local variables with the same aggregate types are declared, which are passed to the function.

3.1.3 Structures and Unions. Figure 7 shows an example of a program including structures and unions generated by this method. As declared in lines 1–13, members of the structures/unions may be any of a scalar element, an array, a structure, or a union. Arbitrary levels of nested structures/unions are allowed (recursive definition is prohibited, though). One limitation regarding a union is that only the first member may be accessed; a union may be initialized by the first member (as in line 20) and only the first member (`x1.m0`) may be referenced or defined (as in lines 22–23).

3.2 Extension of Control Statements

3.2.1 Outline. As for function calls and *while* statements, restrictions on derivation are imposed so that the run time values of all the variables are kept unique. On the other hand, by making use of the semantic information, generation of infinite *while* loops are avoided.

3.2.2 Function Calls. In order to maintain consistency of the semantic information that each variable has a unique value in a program, we enforce the following restrictions on function calls:

```
 1: struct s0 {
 2:   signed int m0;
 3:   unsigned int m1;
 4: };
 5: union u0 {
 6:   signed int m0;
 7:   struct s0 m1;
 8: };
 9: struct s1 {
10:   signed int m0[2][2];
11:   union u0 m1;
12:   struct s0 m2[2];
13: };
14: void func0(struct s1 a0, long a1) {
15:   int t50 = a0.m0.[0][0] + x43;
16:   if (t50 == -59000) {OK();} else {NG();}
17: }
18: int main(void) {
19:   struct s0 t0 = {64, 33};
20:   union u0 x1= {16};
21:   struct s1 x2 = {{{0, 3}, {2, 8}},
      {66}, {{125, 33}, {64, 666}}};
      ...
22:   func0(x2, x33 + x1.m0);
23:   t0.m1 = x3 + x1.m0 - x2.m2[x3-x4].m1 - x15;
24:   if (t0.m1 == 647) {OK();} else {NG();}
25:   ...
26: }
```

Figure 7: Test program with structures/unions

- A function may be called plural times, but the same function must be called exactly with the same argument values every time.
- A function returns the same value every time.

Figure 8 shows an example of generating definition and calls of a function. In the upper code in the figure, function `f1` is called twice in lines 13–14, with the same arguments 5 and 9. `f1` is defined to always return 5 in line 8. Then this code is transformed into the lower code by expanding the constants into the expressions.

If one of the variables in the expressions is volatile, compilers can not tell that `f1` is called with the same argument values. Similarly, compilers can not recognize that `f1` always returns 5. Then, compilers should do code generation and optimization just like when the expressions evaluate to different values, thus our restriction does not severely impair the quality of the test programs. Of course, with a certain probability, all the variables in a expression happen to be non-volatile. In that case, compilers may attempt optimization making full use of this fact. This means that our tests have also chances to test if such optimization results in correct codes.

In lines 6–7, it is checked if the the values of the arguments are correct, which was difficult in differential testing method.

3.2.3 While Statements. In order to make the value of the variable unique, we only generate *while* statements whose iteration count is 0 or 1.

Figure 9 (a) shows how a *while* statement of no iteration is generated. It starts from a *while* statement whose loop condition is constant 0. The constant is expanded into an expression and statements are recursively generated in the loop body. Arbitrary combination of nesting of *while* loops as well as *for* loops and *if* statements are possible. A *while* statement of a single iteration is generated from the template in (b), where the loop condition is 1 and the break condition is 1.

If volatile variables are contained in the loop condition or the break condition, compilers can not tell how many times the loop is iterated. Since they have no choice but to generate codes for general cases, this limitation does not severely degenerate the quality of the test.

```
 1: int f1 ( unsigned short a1, long a2 ){
 2:   t1 = -235;
 3:   t2 = 0;
 4:   ...
 5:   t99 = 234;
 6:   if( a1 == 5 ){OK();} else{NG();}
 7:   if( a2 == 9 ){OK();} else{NG();}
 8:   return 5;
 9: }
10: int main(void){
11:   int x1 = 34;
12:   ...
13:   int t3 = ((x2 + f1(5,9)) & x8) % t2;
14:   int t4 = (t3 < x7) >> f1(5,9) / x5;
15:   ...
```

↓

```
 1: int f1 ( unsigned short a1, long a2 ){
 2:   t1 = ((x23 * a1) >> x2 & x8);
 3:   t2 = (t1 | x5);
 4:   ...
 5:   t99 = (x15 * x2) - (x10 - a1);
 6:   if( a1 == 5 ){OK();} else{NG();}
 7:   if( a2 == 9 ){OK();} else{NG();}
 8:   return (a1 - x3) * (t7 << (x3 / t11));
 9: }
10: int main(void){
11:   int x1 = 34;
12:   ...
13:   int t3 = ((x2 + f1((x2*x7), (x1-x5))) & x8) % t2;
14:   int t4 = (t3 < x7) >> f1((t3/x9), (x8-x7)) / x5;
15:   ...
```

Figure 8: Function definition and calls

```
while( 0 ) {
  <statements>
}
```
→
```
while( ((x3<<t7)&(a8%x5)) ) {
  <statements>
}
```

(a) *While* loop of no iteration

```
while( 1 ) {
  <statements>
  if ( 1 ) {break;}
}
```
→
```
while( (t2*x4)!=(x1>>x10) ) {
  <statements>
  if ( (a1-x11)-(x2/t5) ) {break;}
}
```

(b) *While* loop of a single iteration

Figure 9: Generation of *while* statement

```
switch ( 4 ) {
  case 3:
    <statements>
    break;
  case 4:
    <statements>
    break;
  default:
    <statements>
    break;
}
```
→
```
switch ( (t2*x4)!=(x1>>x10) ) {
  case (a0-x13)-(x21/t3):
    <statements>
    break;
  case (x3-a3)*(t6*x12):
    <statements>
    break;
  default:
    <statements>
    break;
}
```

Figure 10: Generation of *switch* statement

3.2.4 Switch Statements. Figure 10 shows how a *switch* statement is generated in our method. At first, a set of *case* labels ({2,9,13} in this example) is randomly generated, one of which (13) is used for the default clause. Then one of them is chosen randomly as the value of the *switch* expression. A prototype of a *switch* statement is generated, as on the left side of the figure, and then the constants are expanded into expressions.

3.3 Minimization

The minimization of error programs can be realized by adding reduction rules regarding the control sentences and the data types introduced in this paper. For example, with respect to function calls, new rules include substitution of a function call with its return value, deletion of the definition of a function that are not called at all. As for the *while* statements, examples of transformations are removal of a while loop whose body is empty, deletion of all the statements in the body of a loop whose iteration condition is 0, replacing a *while* loop which iterates only once with the statements in its body.

When an error program contains an array, a structure, or a union, it is necessary to confirm whether the error is really coming from them. For this purpose, reduction of the elements of these compound data types into flat scalar variables is attempted. Figure 11 shows an example of minimizing array variables. An array variable `t0` is declared in line 1 and its elements are referenced in line 5. If this program exhibits an error, the array elements are replaced by scalar variables to see if the error vanishes. The same reduction is also attempted on structures and unions.

```
1: int t0[3] = {0, 1, 2};
2: x0 = 5;
3: int main(void)
4: {
5:   t1 = t0[0] + x0 + t0[1 * t0[1]];
6:   if (t1 == 6) {OK();} else {NG();}
7: }
```

↓

```
1: int t0_0 = 0; int t0_1 = 1;
2: x0 = 5;
3: int main(void)
4: {
5:   t1 = t0_0 + x0 + t0_1;
6:   if (t1 == 6) {OK();} else {NG();}
7: }
```

Figure 11: Minimization on array variables

Table 1: Comparison with Csmith

	Csmith	Orange4 ("√*": new features introduced in this paper)
if	√	√
for	√	√ + General expressions for loop bounds − Same values assigned to variables every iteration
function	√	√* + Argument values are checked
while	−	√* − Only 0 or 1 iteration
switch	−	√*
goto	√	−
array	√	√* + Complex expression in subscripts + Variable-length arrays
structure	√	√* + Member of structures may be arrays of structures
bit field	√	−
pointer	√	−

4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

Orange4, a random test system for C compilers, is extended based on the proposed method in this paper. It is written in Perl5 and runs on Unix environment such as Ubuntu Linux, Cygwin, Mac OSX, etc.

Table 1 is a summary of the C program constructs generated by Csmith and Orange4 (where "√*" indicates new features introduced in this paper). Csmith has still advantages in terms of syntactic broadness, for it covers pointers and bit fields, particularly.

```
 1: int a[2] = {0,1};
 2: int x = 129;
 3: int main (void)
 4: {
 5:   volatile int v = 0;
 6:   int t = x;
 7:   for (int i=0; i < (1+v+v+v+v+v+v+v+v+a[a[0]]); i++) {
 8:     t = a[(signed char)(130-x)];
 9:   }
10:   if (t != 1) __builtin_abort();
11:   return 0;
12: }
```

The final value of t should be 1, but compiled with with -O3 option, the check in line 10 failed.

Figure 12: Minimized error program for GCC-8.0.0

```
1: #define INT_MAX 0x7FFFFFFF
2: char a[1][1] = { {1} };
3: int x = 0;
4: int y = -INT_MAX;
5: int main (void)
6: {
7:   if (a[x][INT_MAX+y] != 1) __builtin_abort();
8:   return 0;
9: }
```

With -O1 option, compilation failed with an error message "linker command failed with exit code 1."

Figure 13: Minimized error program for LLVM-6.0

Table 2: Time for random test

compiler	Csmith		Orange4 (extended)	
	time [h]	#error	time [h]	#error
GCC-4.4.7	46.4	5	54.5	25
GCC-4.5.4	30.2	1	56.7	30
GCC-4.6.4	31.3	0	56.4	10
GCC-4.7.3	35.6	0	62.1	10

#test: 100,000
Average size of test program: 15.3KB (Csmith), 8.9KB (Orange4)
CPU: Xeon (3.60GHz) with 16GB RAM

On the other hand, extended Orange4 is better in quality in some constructs, especially in the array subscripts.

With the extension proposed in this paper, we have successfully detected two new bugs in the latest version of GCC and LLVM, for which Csmith and previous Orange4 can not generate the test cases.

Figure 12 is an error program that detected a bug in GCC-8.0.0 (latest development version at the time of the experiment). The program is a result of hand minimization after automatic minimization by Orange4. The *for* loop in lines 7–9 iterates just once and the expected value of t is 1, but with -O3 option the check in line 10 failed (the observed value of t was 0). The error vanishes if either of the references to the array elements in line 7 is replaced by that to a scalar variable. The error was reported to the developer through Bugzilla[3] and the bug was fixed.

Figure 13 is a minimized error program for LLVM-6.0 (latest at the time of the experiment). Compiling with -O1 option results in a "linker command error" and compilation failed. To reproduce the same error, reference to the 2-dimensional array with the expression in the second subscript was necessary. The error was reported to the Bugzilla of LLVM[4].

[3]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=83580
[4]https://bugs.llvm.org/show_bug.cgi?id=35159

Table 3: Time for minimization

Error program	time [sec]	
	C-Reduce	Orange4
#01	328.2	6.7
#02	155.0	24.8
#03	990.6	15.7
#04	106.0	13.9
#05	254.3	31.3
#06	76.1	4.2
#07	1697.3	5.7
#08	3888.2	23.2
#09	1521.5	20.4
#10	106.8	4.5
#11	72.8	6.6

CPU: Xeon (3.60GHz) with 16GB RAM

Table 2 shows the time necessary for testing GCCs with 10,000 random programs. Time is measured on Xeon (3.60GHz) with 16GB RAM. The size of the random programs was targeted to about 1,000 lines both for Csmith and Orange4, but they fluctuated because of the randomness. The average size of the test programs resulted in 15.3KB for Csmith and 8.9KB for Orange4. Orange4 was slower than Csmith but it took less than twice hours. Since the same sequences of test programs are generated in both Csmith and Orange4, the difference in test hours seems to come from compilation time. The columns "#error" show the numbers of detected errors but these figures are just for reference, for Csmith had already detected many bugs in GCCs of version 4.3.x and 4.4.x which had been fixed.

Table 3 lists the CPU time for minimizing the 11 error programs detected by the proposed method on GCC-4.8.5. Orange4 completed minimization within a minutes for all the programs. C-Reduce took more time because it is a general minimizer while Orange4 starts from the error programs' ASTs which are annotated with semantic information; it can use the information to avoid transformation that triggers undefined behavior.

5 CONCLUSION

We have proposed an extension of the equivalent transformation based random program generation method for automated testing of C compilers. This extension enables generation of test programs with *while* and *switch* statements and array accesses with complex expressions, which have not been done by the existing methods. With this feature, extended Orange4 have detected bugs in the latest version of GCC and LLVM.

We are now still working on the extension of Orange4 to fill the gap between Orange4 and Csmith. Besides validation test of compilers, we are also interested in performance test of compilers [4, 6] and we are planning to apply extended Orange4 to performance test. It is also a future work to develop random test generator for other languages than C, such as Java. It is relatively easy to generate Java codes from ASTs of Orange4, but we think the current framework of Orange4 is insufficient for covering broad language constructs of Java such as a class, inheritance, etc.

Orange4 is available on GitHub since December 22, 2016 [5].

[5]https://github.com/ishiura-compiler/Orange4

ACKNOWLEDGMENTS

Authors would like to thank all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion and advice on this research. This work was partly supported by JSPS Kakenhi Grant #25330073.

REFERENCES

[1] Free Software Foundation. [n. d.]. Installing GCC: Testing. Retrieved July 26, 2018 from http://gcc.gnu.org/install/test.html

[2] T. Fukumoto, K. Morimoto, and N. Ishiura. 2012. Accelerating regression test of compilers by test program merging. In *in Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 42–47.

[3] Plum Hall. [n. d.]. The Plum Hall Validation Suite for C. Retrieved July 26, 2018 from http://www.plumhall.com/stec.html

[4] A. Hashimoto and N. Ishiura. 2016. Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs. In *IPSJ Trans. System LSI Design Methodology*, Vol. 9. 21–29.

[5] Y. Hibino, H. Ikeo, and N. Ishiura. 2017. CF3: Test Suite for Arithmetic Optimization of C Compilers (letter). In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E100-A. 1511–1512.

[6] M. Iwatsuji, A. Hashimoto, and N. Ishiura. 2016. Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing (short paper). In *in Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*. 2–3.

[7] V. Le, C. Sun, and Z. Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 386–399. https://doi.org/10.1145/2858965.2814319

[8] V. Le, C. Sun, and Z. Su. 2015. Randomized Stress-Testing of Link-Time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 327–337. https://doi.org/10.1145/2771783.2771785

[9] C. Lindig. 2005. Random testing of C calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05)*. 3–12.

[10] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. In *IPSJ Transactions on System LSI Design Methodology*, Vol. 7. 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

[11] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation. In *Proceedings of Asia and Pacific Conference on Circuits and Systems (APCCAS2016)*. 676–679. https://doi.org/10.1109/APCCAS.2016.7804063

[12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*. 335–346. https://doi.org/10.1145/2345156.2254104

[13] C. Sun, V. Le, and Z. Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. 849–863. https://doi.org/10.1145/3022671.2984038

[14] X. Yang, Y.Chen, E. Eide, and J. Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. 283–294. https://doi.org/10.1145/1993316.1993532