一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2017-87,CPSY2017-131,RECONF2017-75(2018-01)

等価変換に基づく C コンパイラテストシステムにおける制御文生成の強化

岩辻　光功†　　石浦菜岐佐†

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2-1

あらまし　本稿では, テストプログラムの等価変換に基づく C コンパイラのランダムテストシステムにおいて, 制御文の生成機能を強化する手法を提案する. コンパイラのランダムテストの方式は, 差分法と期待値計算に基づく手法に大別される. 前者は, 文法に従って幅広い構文のプログラムを生成できる一方で, 未定義動作の回避が課題になる. これに対し後者では, 未定義動作を回避するようにプログラムを生成できるが, プログラム生成の際に構文木に加えて意味に関するデータ構造を構築する必要があるため, 差分法に比べてプログラムが生成可能な文法の範囲が限定される. 本稿では, 後者のプログラム生成法において生成可能な制御文の種類を増やす手法を提案する. 等価変換によるプログラム生成手法に基づき, 従来から生成可能であった if 文, for 文に加えて, while 文, switch 文, 関数呼び出しを含むプログラムを生成可能にする. このうち, while 文, switch 文は, 差分法では生成が難しかったものである. 本手法に基づくテストシステムを Orange4 に追加実装して実験を行ったところ, GCC-4.4, GCC-4.8 において従来の手法では検出できない不正コード生成を検出することができた.
キーワード　コンパイラ, ランダムテスト, 等価変換

Reinforcing Generation of Control Flow Statements in Random Test System of C Compilers Based on Equivalence Transformation

Mitsuyoshi IWATSUJI† and Nagisa ISHIURA†

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article proposes a method of reinforcing generation of control statements in random testing of compilers based on equivalence transformation on test programs. Existing methods for compiler random testing are divided into two classes, differential testing and oracle-based testing. The former approach can generate programs with wider syntax constructs but has difficulty in avoiding undefined behavior. The latter, on the other hand, can generate programs without undefined behavior but the programs are with less syntax variety, because it must construct data structures to hold semantic information while generating abstract syntax trees. This article attempts to strengthen the latter method so that it can generate a wider variety of control statements. A test generation method based on equivalence transformation on test programs is extended to generate *while* statements, *switch* statements, and function calls, in addition to *if* and *for* statements. An enhanced version of the test system Orange4 based on the proposed method has generated error programs to detect bugs in GCC-4.4 and GCC-4.8, which could not be generated by the existing methods.
Key words　compiler, random testing, equivalence transformation

1. はじめに

コンパイラはソフトウェア開発の基盤ツールであり, 高い信頼性が求められる. コンパイラの不具合はソフトウェアの品質に致命的な影響を与えるため, コンパイラには徹底的なテストが求められる.

コンパイラのテストは膨大な数のテストプログラムから成るテストスイート [1]〜[3] を用いて行われるが, テスト数が有限である以上, 不具合の見逃しは避けられない. テストスイートによるテストを補う手法として, ランダムに生成したプログラムを用いて対象のコンパイラをテストするランダムテスト [4] が行われる.

コンパイラのランダムテストでは, ランダムに生成したプログラムをコンパイル・実行して得られる結果が正しいことを

いかにして判定するか, および, ゼロ除算等の未定義動作を含むプログラムの生成をいかにして避けるかの 2 点が課題になる. これまでに提案されているランダムテストの手法は, 差分法 (differential testing) [5] に基づくものと, 期待値計算に基づくもの [6] [7] に大分される.

差分法による手法では, ランダムに生成した 1 つのテストプログラムを複数のコンパイラ (あるいは同一コンパイラの複数のバージョン) でコンパイルし, その実行結果を比較することによってコンパイルの誤りを検出する. 差分法に基づく C コンパイラのランダムテストシステム Csmith [8] は, C 言語の広範な文法をカバーしており, 2010 年までの 3 年間に GCC および LLVM/Clang の最新版において, 新しい不具合をそれぞれ 79 件と 202 件検出している. Proteus [9], Athena [10], Helmes [11] は, 既存のテストプログラムに対して出力を変化させないような変換を行って新たなテストケースを生成する手法であるが, やはり差分法に基づいて実行結果の正誤判定を行っている.

差分法では, 文法に従って幅広い構文のプログラムを生成できる一方で, 未定義動作の回避が課題になる. Csmith では文法やプログラムの生成規則に制限を加えることにより未定義動作を回避しているが, これによって生成できるテストプログラムはどうしても未定義動作に関して悲観的なものに制限されてしまう.

これに対し, 期待値計算に基づく方法では, ランダムプログラム生成と同時にそのプログラムの正しい実行結果 (期待値) を計算し, これを用いて実行結果の正誤判定を行う. この方法では, プログラムの生成過程で未定義動作の可能性を検出した場合に, それを回避するようにプログラムを構築できるため, 自由度の高いプログラムを生成できる. Orange3 [6] では C プログラムの算術式の生成時に未定義動作があればそれを回避するように式を修正する方法により, Csmith では検出できなかった GCC, LLVM の不具合を検出している. また, Orange4 [7] では, プログラムの等価変換に基づいて, 未定義動作が生じないプログラムだけを生成している.

しかし, 期待値計算に基づく手法では, プログラム生成の際に構文木だけでなく意味に関するデータ構造も構築する必要があり, 差分法に比べてプログラムが生成可能な文法の範囲が限定されてしまう. Orange3 では for 文を [12], Orange4 では for 文と if 文を生成プログラムに加えているが, それ以上の制御文は扱えていない. 期待値計算に基づく方法において, 制御文を増やせれば, 意味情報を生かして差分法では生成できないテストプログラムを生成できると考えられる.

そこで本稿では, Orange4 のプログラム生成方法を拡張することにより, 期待値計算に基づく C コンパイラのランダムテストシステムにおいて, 生成可能な制御文の種類を増やす手法を提案する. 具体的には, if 文, for 文以外に while 文, switch 文, 関数呼び出しを追加する. while 文, switch 文は Csmith では生成できていなかったものである. また, 意味情報を利用して, 無限ループや無限再帰呼出しが起こらないようにプログラムを生成することができる. 本手法に基づくランダムテストシステムを Orange4 に追加する形で実装した結果, GCC-4.4, GCC-4.8 において, これまでの手法では検出できない不正コード生成を 2 件検出することができた.

2. コンパイラのランダムテスト

2.1 ランダムテストの流れ

コンパイラのランダムテストは, 図 1 に示すように, ランダムに生成したテストプログラムをテスト対象のコンパイラでコンパイルし実行するという処理を, 時間の許す限り実行するというものである. もし, コンパイラがクラッシュしたり, プログラムの実行結果が正しくない等のエラーが検出されれば, そのプログラム (エラープログラムと呼ぶ) を保存して分析する.

テストプログラムは数千行規模に及ぶことがあるため, そのような規模のエラープログラムが与えられてもエラーの原因の特定は難しい. そこで, エラープログラムを縮約し, エラーの起こるできるだけ小さなプログラムを求める処理が行われる. この処理は, エラープログラムの最小化と呼ばれ, コンパイラのランダムテストにおいて重要な位置を占める.

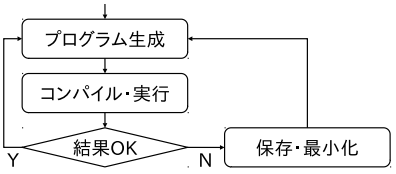

図 1　ランダムテストの流れ

2.2 Csmith

Csmith [8] は差分法 [5] に基づく C コンパイラのランダムテストシステムである. 関数呼び出し, for 文等の制御文に加え, 配列や構造体, ポインタ等を含む C 言語の幅広い構文要素を含むテストプログラムの生成が可能である.

Csmith のテストプログラムの例を図 2 に示す. 制御文に関しては, if 文, for 文, 関数呼び出しが生成される. プログラム生成時には実行時の変数の値はわからないため, プログラムは未定義動作を引き起こさないように工夫されている. 例えば, 25 行目の safe_mul_func_uint8_t_u_u は乗算を行うマクロであるが, オペランドを検査してオーバーフローやアンダーフローが起きない場合には乗算結果を, そうでなければ第 1 オペランドを値とするようになっている. また, 21 行目では, for 文が無限ループに陥らないよう, バウンドは定数のみとしている.

Csmith では, 図 2 には現れていない goto 文も生成されるが, while 文や switch 文は生成されない. これは, 実行時の式の値がわからないと, while 文が無限ループに陥ったり, switch 文の制御式の値が case 節のどれとも一致しない確率が高くなるのを回避できないことが理由と考えられる.

Csmith で検出したエラープログラムの最小化には, 汎用最小化ツールである C-Reduce [13] が使用される. C プログラムのソースコードとエラーを判定するコマンドラインを与えると, エラーが起こる最小のプログラムを出力する. C プログラムを縮約する種々の変換を実装しているが, その適用によって未定義動作が引き起こされることがあるため, 静的解析ツールを用いてそのような変換の適用を排除している. 縮約による未定義動作が頻発する場合には, 最小化に長時間を要し, 最小化に失敗することがある.

2.3 Orange4

Orange4 [7] は, 期待値計算に基づく C コンパイラのランダム

```
...
01: static uint32_t * func_65(uint32_t * p_66, uint8_t  p_67,
02: uint32_t * p_68, uint16_t  p_69)
03: { /* block id: 17 */
04:   int32_t **l_93[8] =
05:   {&g_84, &g_84, &g_84, &g_84, &g_84, &g_84, &g_84, &g_84};
06:   uint16_t l_98[3];
07:   uint8_t l_127 = 3UL;
08:   int i;
09:   for (i = 0; i < 3; i++)
10:     l_98[i] = 0xB5CEL;
11:   if (func_2(g_31[3]))
12:   { /* block id: 18 */
13:     int32_t **l_88 = &g_84;
14:     uint16_t *l_96[1];
15:     int8_t *l_99 = (void*)0;
16:     int8_t *l_100 = &g_101;
17:     int32_t l_102 = 0x4F7977D3L;
18:     int32_t *l_103 = &g_85;
19:     int32_t *l_107 = &g_108;
20:     int i;
21:     for (i = 0; i < 1; i++)
22:       l_96[i] = &g_97;
23:     (*l_88) = p_68;
24:     l_102 = ((((safe_lshift_func_uint16_t_u_u((
25:     safe_mul_func_uint8_t_u_u(((**l_88) & (((*p_68) , p_67) &&
26:     ((((((func_2((**l_88)), (void*)0) == (void*)0) > (l_93[7] !=
27:     (((*l_100) = func_2((safe_rshift_func_uint16_t_u_u((g_97 ^=
28:     65535UL), l_98[1]))))), &g_84))) < g_29) != p_67) > g_77))),
29:     p_67)), p_67)) , 0x479FL) , p_67) || 0x6366C4368EC64C76LL);
30:     l_103 = func_70(p_69, (*g_84), p_67, p_68, (*l_88));
31:     g_85 ^= ((((0xDEA6L != (((2UL == (safe_add_func_uint32_t_u_u(
32:     4294967288UL, (p_69 | p_67)))), ((g_106, ((*l_107) = (&l_103
33:     != &l_103))) , (safe_mul_func_uint8_t_u_u(p_69, 0x7CL)))) ,
34:     p_67)) > g_21[3][4]) ^ (*p_66));
35:   }
      ...
36:   return &g_21[7][5];
37: }
...
```

図 2　Csmith のテストプログラムの例 (一部)

テストシステムである. コンパイラの算術最適化を主な対象にしているが, if 文と for 文を含むプログラムも生成する. Orange4 が生成するテストプログラムの例を図 3 に示す. 27–30 行目に算術式や制御文を含む文が生成されている. 実行時の各変数の値はプログラム生成時に既知であるため, それを用いて 31–32 行目で実行結果の正誤判定を行っている. また, 27 行目の for 文のバウンドにも定数ではなく式を生成している.

Orange4 では, 図 4 に例を示すように, 等価変換により定数から複雑な算術式を生成する. まず最初に式の値を決定し, それを値が同じになるような式に展開するという変換を繰り返すことにより式を生成する. この際, オペランドを適切に選択することにより未定義動作が発生しないようにできる.

Orange4 では, プログラム中の各文は高々 1 回しか実行されないという制約を設けている. この制約により, プログラム中で参照される変数および式の値は一意に定まる. Orange4 はプログラムの解析木表現と共に, 全ての変数, 式, および部分式に対してこの値をプログラムの意味情報として保持している.

この制約を満たすために, for 文の生成においては, ループ回数は 0 回または 1 回になるように制御変数のバウンドの式の値を設定している. コンパイラは, 式の値が定数に畳み込める場合にはループを削除する最適化を行うが, 式中に volatile 変数が生成されていて式を定数に畳み込めない場合には, ループ回数が 2 回以上の場合と同じコード生成を行うため, この制約が不具合検出能力を著しく損なうことはないと考えられる.

Orange4 では, エラープログラムの最小化機能を組み込みで実装している. 一般的な最小化手法 [14] と同様, エラープログラムに対する縮約変換を, それ以上どの変換を適用してもエラーが消失するという状態まで繰り返す. 具体的な縮約変換は

- 代入文を削除する (代入される値を被代入変数の初期値とする)
- 変数を定数値に置き換える
- 定数値同士の演算を結果となる値で置き換える
- 参照されない変数を削除する

等である. これらの変換は, 意味情報 (変数と部分式の値) を持った解析木を入力として行われるため, プログラムの縮約に際して未定義動作を引き起こすことがない. このため, 一般のプログラムを入力とすることはできないが, C-Reduce よりも高速であり, 未定義動作の頻発によって最小化が失敗することもない.

一方で, Orange4 では, プログラム生成のために解析木と共に意味情報を構築しなければならず, また生成されるプログラムは参照される変数の値が一意に定まるという制約を満たさなければならないため, Csmith に比べるとカバーできる文法の範囲が狭くなっている. 表 1 は, Csmith と Orange4 がテストできる C の構文を比較したものである. Orange4 では, スカラ変数以外のデータ構造に対応しておらず, また制御文に関しても goto 文や関数呼び出しには対応していない.

```
01: #include <stdio.h>
02: #define OK()  printf("@OK@\n")
03: #define NG(test,fmt,val)  printf("@NG@ ("test" = " fmt ")\n",val)
04:
05: const volatile signed char x13 = 127;
06: const volatile signed char x14 = -128;
07: signed long long x17 = 4132859LL;
08: const signed char x18 = -98;
09: const volatile signed long x22 = 3432268666822104507L;
10: static signed char t0 = 0;
11: const volatile signed int x23 = 165236596;
12: unsigned long t1 = 726320268520LU;
13:
14: int main (void)
15: {
16:   volatile signed char x0 = 50;
17:   signed short x5 = 0;
18:   volatile signed short x6 = 98;
19:   const signed long long x8 = -7320LL;
20:   signed long x11 = -1L;
21:   const signed int x12 = 185091;
22:   unsigned long long x16 = 27973123LLU;
23:   static const signed long long x21 = -6864537333670614482LL;
24:   const volatile signed int x24 = 40393715;
25:   signed int i;
26:
27:   for( i = ((signed int)((x12-((signed int)x8))/((signed long)
      (x13+x14)))); i > ((signed int)(((signed long long)(x16^x17))
      *((signed long long)((signed long)(x18/x6))))); i +=
      ((signed int)(((signed long)(((signed char)x11)+x21))%
      (x22/((signed short)x11))))) {
28:     t0 = ((signed char)(((signed int)((signed char)(((signed int)
        x0)<<((signed long long)x5))))>>((signed long)((signed short)
        (((unsigned short)x5)&&((unsigned long)x22))))));
29:     t1 = ((unsigned long)(((signed long)((signed short)((
        (signed long)x5)/((signed long)x21))))|((signed long)
        ((unsigned long)(x23%x24)))));
30:   }
31:   if (t0 == 50) { OK(); } else { NG("t0", "%d", t0); }
32:   if (t1 == 3661736LU) { OK(); } else { NG("t1", "%lu", t1); }
33:   return 0;
34: }
```

図 3　Orange4 のテストプログラムの例

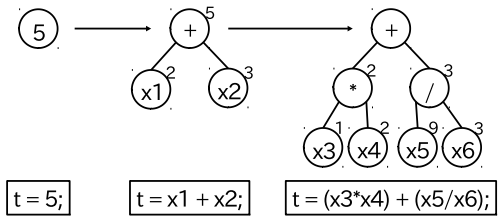

図 4　Orange4 の式生成

3. Orange4 の制御文生成の強化

本稿では, 等価変換に基づきテストプログラム内に関数呼び出し, while 文, および switch 文を生成する手法を提案する. 関数呼び出しと while 文に関しては, いくつかの制限を課すことによりプログラム中で参照される変数の値が一意に定まるようにする. 逆に, プログラム中の式の値が生成時にわかること

表 1　Csmith と Orange4 の違い

	Csmith	Orange4	提案手法
for	○	○	○
if	○	○	○
関数呼び出し	○	×	○
while	×	×	○
switch	×	×	○
goto	○	×	×
配列	○	×	×
構造体	○	×	×

を生かし, 無限ループを引き起こす while 文や case 節が選択されない switch 文の生成を避けることができる.

3.1　関数呼び出し

プログラム中で参照される変数の値が一意に定まるようにするため, 本稿の手法で生成する関数は, 複数回呼び出されても良いが, 呼び出し時の引数の値は毎回同じであり, 関数の返り値も毎回同じ, という制約を課す.

図 5 に関数の定義と呼び出しの原型 (上側), およびそこから式の展開により生成されるコード片 (下側) の例を示す. 8 行目で f1 の返り値は定数 (5) とし, 14–15 行目の関数呼び出しでは f1 を同じ値の引数 (5, 9) で呼び出している. 一旦このような原型を生成した後, 返り値と引数は Orange4 の式展開の機能を使って下側のように同じ値を持つ式に変換する. 式中に volatile 変数が含まれていれば, コンパイラは同じ引数で呼び出されていることや, 同じ値を返していることは判定できない. 6–7 行目では, 引数の値が生成時に既知であることを利用して渡された引数の値を正しいかどうかを検査しているが, これは差分法ではできなかったことである.

関数呼び出しに関しては, 関数呼び出しの深さ, および, グローバル変数の参照のタイミングに注意が必要である.

無限再帰呼出しを避けるため, 本稿の生成法では,

- 関数にシリアルな番号を付し
- 自分より若い番号の関数しか呼び出さない

という制約を設ける. さらに, 関数呼び出しの深さが増えるとプログラムの実行時間が増えてテストの効率が下がるため, 生成する関数の数に上限を設ける.

関数中でグローバル変数を参照している場合には, その参照時点までにそのグローバル変数を定義している関数が実行されている必要がある. これは,

- 各関数 f について, その関数内で定義されるグローバル変数のリスト $G(f)$ を作成しておき,
- 各関数 g の定義を生成する際に, ある関数 f を呼び出す毎に $G(f)$ のグローバル変数を, g が参照可能な変数のリストに追加する,

という処理により実現できる.

3.2　while 文

差分法で while 文を生成すると, 継続条件式の値が制御できないため, 無限ループになる可能性が高い. これに対し, Orange4 の生成法では, 評価値が指定通りの式を生成できるため, この問題を回避できる. ただし, 参照される変数の値を一意にするため, while 文でも for 文と同様, ループの繰り返し回数を 0 か 1 に限定する.

繰り返しの回数が 0 の while 文は, 図 6 (a) のように, 継続条件が 0 の while 文の原型から開始して, 継続条件を等価な式

```
01: int f1 ( unsigned short a1, long a2 ){
02:   t1 = -235;
03:   t2 = 0;
04:   ...
05:   t99 = 234;
06:   if( a1 == 5 ){OK();} else{NG();}
07:   if( a2 == 9 ){OK();} else{NG();}
08:   return 5;
09: }
10:
11: int main(void){
12:   int x1 = 34;
13:   ...
14:   int t3 = ((x2  f1(5,9)) & x8) % t2;
15:   int t4 = (t3 < x7) >> f1(5,9) / x5;
16:   ...
17: }
```

↓

```
01: int f1 ( unsigned short a1, long a2 ){
02:   t1 = ((x23 * a1) >> x2 & x8);
03:   t2 = (t1 | x5) % (a2 && t1);
04:   ...
05:   t99 = (x15 * x2) - (x10 - a1);
06:   if( a1 == 5 ){OK();} else{NG();}
07:   if( a2 == 9 ){OK();} else{NG();}
08:   return (a1-x3);
09: }
10:
11: int main(void){
12:   int x1 = 34;
13:   ...
14:   int t3 = ((x2  f1((x2*x7), (x1-x5))) & x8) % t2;
15:   int t4 = (t3 < x7) >> f1((t3/x9), (x8-x7)) / x5;
16:   ...
17: }
```

図 5　関数宣言文と呼び出し文の例

に展開することにより生成する. 文リストにはランダムな文のリストを再帰的に生成する. 同様に, 繰り返し回数が 1 の while 文は, 図 6 (b) のように, 継続条件と中断条件を 1 とした while 文の原型から生成する. 継続条件や中断条件の式が定数畳み込みで 0 や 1 に縮約される場合はループを削除する最適化が行われるが, 条件式中に volatile 変数が含まれている場合には, 繰り返し回数が 2 以上の場合と同じ条件式で最適化が行われる.

なお, コンパイラのコード生成の誤りのために, これらのループが無限ループに陥る場合もあるため, 1 プログラムの実行時間に上限を設け, その時間内に実行が終わらない場合には強制終了してエラーとする等の工夫が必要になる.

3.3　switch 文

プログラム生成時に式の値を制御できることを利用すれば, 所望の case 節に分岐する switch 文が容易に生成できる.

本手法で生成する switch 文の原型を図 7 に示す. まず, switch 文中に生成する case 節のラベルの集合 (この例では {3, 4}) を決定し, switch 文の選択式 (この例では 4) をこの集合および他の適当な値 (default 節への分岐用) からランダムに選択する. この情報を元に構成した図 7 のような原型から, 式展開と文の挿入を繰り返して switch 文を生成する.

3.4　最　小　化

エラーを検出したプログラムの最小化は, 本稿で追加した制御文に関する縮約の変換を追加することにより実現する. 2.3 節で述べた式の縮約操作や文の削除に加え, 各構文要素に対して以下の変換を可能な限り (それ以上どの変換を適用してもエ

```
while( 0 ){
  文リスト
}
```

↓

```
while( ((x3<<t7)&(a8%x5)) ){
  文リスト
}
```

(a) ループ回数が 0 の while 文

```
while( 1 ){
  文リスト
  if ( 1 ) { break; }
}
```

↓

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

(b) ループ回数が 1 の while 文

図 6　while の生成例

```
switch ( 4 ) {
  case 3:
    文リスト
    break;
  case 4:
    文リスト
    break;
  default:
    文リスト
    break;
}
```

図 7　switch の生成例

ラーが消失するという状態まで) 繰り返す.

関数呼び出し

- 関数呼び出しを返り値で置き換える
- 一度も呼ばれていない関数の定義を削除する

switch 文

- 実行されない case 節を削除する
- 実行される case 節の文のみを残して switch 文を削除する

while 文

- ループ本体が空の while 文を削除する
- 繰返し回数 0 の while 文を削除する
- 繰返し回数 1 の while 文を, ループ本体の文のリストだけを残して削除する

4. 実装と実験結果

提案手法に基づくランダムテストシステムを Orange4 に追加する形で Perl 5.20.2 で実装した. 本システムは Ubuntu Linux 等で動作する.

GCC-4.8, GCC-4.4, LLVM/Clang-3.0, LLVM/Clang-3.9 に対して実験を行った結果を表 2 に示す. 1 プログラム当たりの規模は 800 から 1,000 行程度であり, それぞれのコンパイラで 24 時間テストを実行した. “#test” は生成したテストプログラムの数, “#error” は検出したエラーの数である.

GCC-4.4.7 で検出したエラー 14 件のうち, 今回追加した制御文によるエラーは 10 件であり, 関数呼び出しと switch 文が原因のエラーが 4 件, 関数呼び出しのみが原因のエラーが 6 件だった. また, GCC-4.8.5 で検出したエラー 11 件のうち, 今回追加した制御文によるエラーは 1 件であり, while 文が原因であった.

表 2　提案手法での実験結果

コンパイラ	#test	#error
GCC-4.4.7	32,229	14
GCC-4.8.5	36,744	11
LLVM/Clang-3.0	37,124	17
LLVM/Clang-3.9	31,252	0

24 時間 (CPU Intel(R) Xeon(R) E3–1276 v3 3.60GHz, RAM 15GB)

図 8 は GCC-4.4 で検出したエラープログラムを最小化したものである. 関数呼び出しと switch 文が含まれており, これらを削除するとエラーは消失する.

図 9 は GCC-4.8 を対象に, 生成する制御文を switch 文のみに絞ってテストした際に検出したエラープログラムを最小化したものである. switch 文を削除し, case 節内の文だけを残すとエラーは消失する.

いずれのエラープログラムも従来の Orange4 および Csmith では検出することはできないものである.

```
01:#include <stdio.h>
02:#include <stdarg.h>
03:#define OK()  printf("@OK@\n")
04:#define NG(test,fmt,val)
   printf("@NG@ ("test" = " fmt ")\n",val)
05:void func0(signed int args_num, ... ) {
06:  va_list args;
07:  signed long a1;
08:  signed int a2;
09:  signed long a3;
10:  va_start( args, args_num );
11:  a1 = va_arg( args, signed long );
12:  a2 = va_arg( args, signed int );
13:  a3 = va_arg( args, signed long );
14:  va_end( args );
15:}
16:int main (void) {
17:  volatile signed long x0 = 1;
18:  volatile unsigned short x1 = 1U;
19:  volatile signed char x2 = 1;
20:  volatile signed char x3 = 0;
21:  volatile static unsigned long x4 = 1;
22:  signed long t0 = 1;
23:  unsigned long t1 = 1L;
24:  if( x4||x3 ) { ; }
25:  else { t0 = x1; }
26:  if (t1 == 1LU) { OK(); } else { NG("t1", "%lu", t1); }
27:  if (t0 == 1L)  { OK(); } else { NG("t0", "%ld", t0); }
28:  func0(3, 1, 1, 2);
29:  unsigned long t2 = 1/x2;
30:  switch( x2&&x0 ) { ; }
31:  if (t2 == 1LU) { OK(); } else { NG("t2", "%lu", t2); }
32:  return 0;
33:}
```

図 8　GCC-4.4 で検出したエラープログラム

表 3 は, GCC-4.8.5 に対して提案手法でテストを行った際に検出した 11 件のエラープログラムを Orange4 と C-Reduce で最小化するのに要した時間である. Orange4 では, 1 つのプログラムの最小化は約 300 秒で完了するのに対し, C-Reduce では 1,800 秒で完了しないものがほとんどであった. これは, Orange4 が未定義動作を引き起こさない縮約のみを適用しているためと考えられる.

```
01:int main (void) {
02:  static unsigned int t1 = 1;
03:  volatile int x1 = 1;
04:  switch( x1 ){
05:  case 1:
06:    t1 = 1U;
07:    break;
08:  }
09:  long long t3 =
     (-1LL / t1) - (-0x7FFFFFFFFFFFFFFFLL - t1);
10:  if (t3 != 0x7FFFFFFFFFFFFFFFLL) { __builtin_abort(); }
11:  return 0;
12:}
```

図 9　GCC-4.8 で検出したエラープログラム

表 3　最小化の実行時間の計測

	CPU time (sec)	
	Orange4	C-Reduce
#01	34.8	> 1,800
#02	291.4	> 1,800
#03	48.5	> 1,800
#04	64.1	> 1,800
#05	57.1	> 1,800
#06	307.8	302.4
#07	79.0	> 1,800
#08	112.5	> 1,800
#09	124.6	> 1,800
#10	71.7	> 1,800
#11	12.8	> 1,800

5. むすび

本稿では, C コンパイラのランダムテストシステム Orange4 において, 制御文生成を強化する手法を提案した. 実験の結果, GCC-4.8, GCC-4.4 において従来の Orange4 では検出できない不具合を検出できた. 今後の課題としては, 表 1 にあるような未対応の C の構文への対応や, 関数呼び出しの引数にスカラ変数以外も指定できるようにすることが挙げられる.

謝　辞

本研究に関して御助言を頂いた, 藤原大輔氏 (現 NTT コミュニケーションズ株式会社) に感謝致します. また, 本研究に関してご協力, ご討議頂いた関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　献

[1] C. Lindig: “Find a Compiler Bug in 5 Minutes,” in *Proc. ACM International Symposium on Automated Analysis-Driven Debugging*, pp. 3–12 (Sept. 2005).

[2] T. Fukumoto, K. Morimoto, and N. Ishiura: “Accelerating Regression Test of Compilers by Test Program Merging,” in *Proc. SASIMI 2012*, pp. 42–47 (Mar. 2012).

[3] Y. Hibino, H. Ikeo, and N. Ishiura: “CF3: Test Suite for Arithmetic Optimization of C Compilers (letter),” in *IEICE Trans. Fundamentals*, vol. E100-A, no. 7, pp. 1511–1512 (July 2017).

[4] 石浦菜岐佐: “コンパイラのファジング,” 電子情報通信学会 Fundamentals Review, vol. 9, no. 3, pp. 188–196 (Jan. 2016).

[5] W. M. McKeeman: “Differential Testing for Software,” in *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107 (Dec. 1998).

[6] E. Nagai, A. Hashimoto, and N. Ishiura: “Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions,” *Trans. SLDM*, vol. 7, pp. 91–100 (Aug. 2014).

[7] K. Nakamura and N. Ishiura: “Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation,” in *Proc. APCCAS 2016*, pp. 676–679 (Oct. 2016).

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr: “Finding and Understanding Bugs in C Compilers,” in *Proc. PLDI 2011*, pp. 283–294 (June 2011).

[9] V. Le, C. Sun, and Z. Su: “Randomized Stress-Testing of Link-Time optimizers,” in *Proc. ISSTA 2015*, pp. 327–337 (July 2015).

[10] V. Le, C. Sun, and Z. Su: “Finding Deep Compiler Bugs via Guided Stochastic Program Mutation,” in *Proc. ACM OOPSLA 2015*, pp. 386–399 (Oct. 2015).

[11] Gergo Barany: “Liveness-Driven Random Program Generation,” in *Proc. LOPSTR 2017* (Oct. 2017).

[12] K. Nakamura and N. Ishiura: “Introducing Loop Statements in Random Testing of C Compilers Based on Expected Value Calculation,” in *Proc. SASIMI 2015*, pp. 226–227 (Mar. 2015).

[13] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison and X. Yang: “Test-Case Reduction for C Compiler Bugs,” in *Proc. PLDI 2012*, pp. 335–346 (June 2012).

[14] A. Zeller and R. Hildebrandt: “Simplifying and Isolating Failure-Inducing Input,” *IEEE Trans. on Software Engineering*, vol. 28, issue 2, pp. 183–200 (Feb. 2002).