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
VLD2019-61,CPSY2019-59,RECONF2019-51(2020-01)

等価ミュータント生成による C コンパイラのテストバリエーションの増強

前田　紘輝†　　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, C コンパイラの自動テストにおいて, 等価ミュータント生成によりテストのバリエーションを増強する手法を提案する. 言語の文法に基づいてテストプログラムを生成する生成ベースの手法に比べ, 既存のプログラムの一部を変異させることによって新たなテストプログラムを生成する変異ベースの手法は, 変異元となるプログラムを適切に選択することによって, その不具合検出能力を向上させたり, 特定機能のテストを重点的に行える等の利点がある. しかし, 既存の変異ベース手法では, テストプログラムが未定義動作を引き起こすのを回避するために, 適用される変異操作が限定され, 生成できるテストのバリエーションが限られるという課題がある. 本稿では, 変異元プログラム中の全ての変数の型と値の情報を把握することによって, 既存の手法では行われていないテストプログラムの変異を提案する. 変数の修飾子・データ型の変異, 変数・配列・構造体・共用体の相互置き換え, および制御文による文のブロック化を, 変異元プログラム中の変数の持つ値が変化しないように行うことによって等価ミュータントを生成する. 本手法に基づくテストシステムを実装し, 過去のコンパイラのエラープログラムを変異元プログラムとして実験を行ったところ. GCC-5.5.0 LLVM/Clang-3.5.2 において従来の変異手法では検出できない不具合を検出した.
キーワード　コンパイラ, 自動テスト, 等価ミュータント

Increasing Test Variation for C Compilers by Equivalent Mutant Generation

Hiroki MAEDA† and Nagisa ISHIURA†

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

Abstract　This article proposes a method of increasing variation of test programs in automatic testing of C compilers by means of equivalent mutant generation. A mutation-based method, where test programs are generated by mutating existing test programs, has different merits from a grammar-based generation method, especially when seed test programs are properly chosen. One of the challenges for the mutation-based method is that applicable mutations have been limited because the mutations must not induce undefined behavior in test programs. This article introduces new types of mutations by bookkeeping the values of all the variables in seed test programs. Our new mutations generate equivalent test programs from a seed program in a sense that all the variable updates are done with the same values both in the seed and mutated programs. The mutations are based on 1) replacement of types and modifiers of variables, 2) interchange among scalar variables, array elements, and struct members, and 3) nesting statements with conditional and loop constructs. A test system based on the proposed method has detected bugs in GCC-5.5.0 and LLVM/Clang-3.5.2 by test programs which can not be generated by existing methods.
Key words　compiler, automated testing, equivalent mutant

1. はじめに

ソフトウェア開発の基盤ツールであるコンパイラには高い信頼性が求められる. コンパイラの不具合に起因するソフトウェアの不具合が発生すると, その不具合を検出できても, 原因がコンパイラにあると特定するまでに時間がかかり, ソフトウェア開発に多大な影響を与える. そのため, コンパイラには信頼性確保のための徹底的なテストが必要となる.

コンパイラのテストは通常数千から数千万のテストプログラムからなるテストスイート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.

テストスイートを経ても潜在する不具合を検出するために, コンパイラの自動テストが行われる. コンパイラの自動テストでは, テストプログラムを自動生成し, 生成したプログラムをコンパイル・実行し, 得られた結果が正しいかどうか検証することによって不具合を検出するという一連の処理を時間の許す限

り行う. コンパイラの自動テストでは, ゼロ除算や配列の範囲外参照などの「未定義動作」を含むプログラムの生成をいかに回避するかが重要な課題となる.

これまでに提案されてきたコンパイラのテストプログラム生成手法は生成ベースの手法と変異ベースの手法に大分される.

生成ベースの手法 [2] [3] [4] では, 言語の文法に基づいてテストプログラムを生成することによって, 複雑なプログラムを生成することができ, 広範なテストを行うことができる.

これに対し既存のプログラムの一部を書き換えることによってテストプログラムを生成する変異ベースの手法 [5] [6] [7] [8] [9] では, 少数の既存のテストプログラムから大量のテストを生成することができ, 適切に元プログラムを選択することによって, その不具合検出能力を向上させたり, 特定機能のテストを重点的に行うことができる. 特に, 修正済みの不具合を引き起こすプログラムを元プログラムとすることによって, その不具合がきちんと修正できているかを重点的にテストできる. SPE [9] は変異ベースのテスト手法であり, 過去の GCC 不具合プログラムが集約されたテストスイートを元プログラムとすることによって 6 ヶ月で 216 件の不具合を検出している.

しかし, これらの手法は変異させたプログラムが未定義動作を引き起こさないことを保証するために変異操作に制限を加えている. これは, プログラムを不用意に書き換えると変数の値が変化して未定義動作を引き起こしてしまう可能性があるからである.

本稿では, プログラム中の全ての変数と式の値を把握することによって, 従来手法では行われていなかったプログラムの変異を行う方法を提案する. 本手法で行う新しい変異は, 変数の修飾子・データ型の変異, 変数・配列・構造体・共用体の相互置き換え, 制御文による文のブロック化である.

本手法に基づくテストシステムを実装した結果, GCC-5.5.0, LLVM/Clang-3.5.2 において, これまでの変異手法では検出できない不正コード生成を 3 件検出することができた.

2. コンパイラの自動テスト

2.1 コンパイラの自動テストの流れ

コンパイラの自動テストの流れを図 1 に示す. テストプログラムをランダムに生成し, 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・実行する. コンパイラのクラッシュや実行結果の誤りなどのエラーを検出すればそのエラープログラムを保存する. この処理を時間の許す限り繰り返す. エラーを検出した場合に, プログラム中のエラーを引き起こす箇所以外の不要部分を削減し, できるだけ小さなプログラムに縮約する「最小化」という処理を行う.

C コンパイラのテストでは, 未定義動作を含むプログラムの生成をいかに避けるかが重要な課題となる. 未定義動作とはゼロ除算, 符号付き整数のオーバーフロー, 配列の範囲外参照等, 実行結果が言語仕様上定義されていない動作のことである. 未定義動作を含むプログラムは, その実行結果がどのようなものであっても言語規格上正しいことになるので, テストプログラムとして意味をなさない. しかし, 未定義動作はプログラムの動的な振る舞いに依存するため, 未定義動作を含むプログラム生成の回避は難しい課題である [1].

2.2 変異ベースのテスト

変異ベースのテストはあるプログラムの一部を変異することによって複数のテストプログラムを生成するものである. プログラムを変異させる操作のことをミューテーションと言い, ミューテーションによって生成されるプログラムをミュータントと呼ぶ.

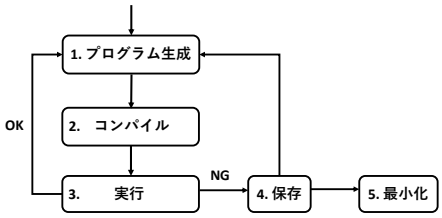


図 1　コンパイラの自動テストの流れ

2.2.1 Equivalence Modulo Inputs (EMI)

EMI とは, 与えられた入力集合に対して同じ結果を出力する, という意味でのプログラムの等価性である. EMI に基づく変異ベース手法は, あるテストプログラムに対して EMI で等価なプログラムを生成し, そのプログラムが元のプログラムと同じ結果を出力するかによってコンパイラのテストを行う. Orion [6], Athena [7], Hermes [8] は EMI に基づくコンパイラのテストシステムであり, コードの挿入・削除に基づき EMI 等価なプログラムを生成する.

Orion は既存プログラム中の実行されないコードをランダムに削除, Athena は既存プログラム中の実行されない部分にコードを挿入することによって新たなテストプログラムを生成する. コードが実行されるか否かはカバレッジツールで判定する. 未定義動作を回避するため, 実行される部分の変異は行なっていない.

Hermes は Orion, Athena の手法に加えて, 実行される部分へコードの挿入を行う. コードを挿入する位置での全ての変数の値を取得しバックアップをする. 取得した情報を元にプログラム中の変数を使用した文を挿入する. 挿入した文の後には, 値が変化した変数を元に戻す処理を追加することによって, 後続の文に未定義動作が発生するのを防いでいる.

しかし, EMI に基づく手法ではコードの挿入・削除は行うが, 変数のデータ型等の変異は行なっていない.

2.2.2 変数配置パターンの列挙

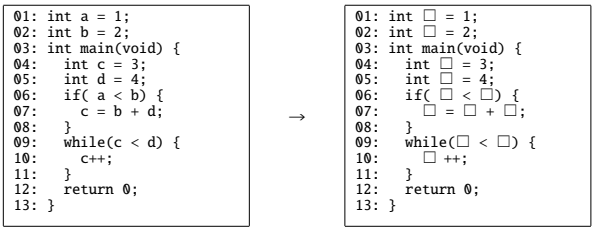
SPE [9] は 10〜30 行程度の小さなプログラムから変数部分を空欄にしたプログラムの骨格を取得し, その骨格に対し全ての変数の配置パターンを列挙することによって複数のテストプログラムを生成する手法である. SPE のテストプログラム生成の例を図 2 に示す. 既存の小さい元プログラム (a) から変数集合 V = {a, b, c, d } と, 変数部分を空欄にしたプログラム骨格 (b) を取得する. 取得したプログラム骨格 (b) に対して変数集合 V の全ての配置パターンを列挙することによって (c) (d) のようなテストプログラムを生成する.

SPE では GCC の過去の不具合が集約されたテストスイート c-tortue (注1) を元プログラムとして使用することによって, 当時の最新版の GCC, LLVM/Clang コンパイラで 6 ヶ月で 217 件の不具合を検出した. この結果から, 過去の不具合周辺には修正しきれていない不具合が潜在しており, 過去のエラープログラム

(注1)：https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html

を変異元プログラムとして利用することは不具合検出に有効であると考えられる.

しかし, 変数の数や配置箇所数が大きくなると変数配置の組み合わせが爆発的に大きくなるので, 数百行以上の大きなプログラムに対しては適用できない. また, 変数の配置の変異によって変数や式の持つ値が変化し, 未定義動作を引き起こすプログラムを生成する場合がある.



(a) 元プログラム　(b) プログラム骨格

(c) テストプログラム 1　(d) テストプログラム 2

図 2　SPE のテストプログラム生成

2.3　変数と式の値を把握したプログラム生成

Orange4 [4] は, 生成ベースの C コンパイラのランダムテストシステムであるが, プログラム中の全ての変数と式の値を保持したデータ構造を用いてプログラム生成を行う点が他の生成ベース手法 [2] と異なる.

Orange4 が生成するテストプログラムの例を図 3 に示す. 6–26 行目が変数宣言, 28–35 行目が算術式や制御文を含む文, 37–41 行目が計算結果を照合する文である.

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 (ループは本体が高々 1 回実行されるように継続条件が生成される). これにより, 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おり, この情報を用いることによって未定義動作を引き起こさないテストプログラムのみを生成している.

3.　等価ミュータント生成によるテストバリエーションの増強

3.1　等価ミュータント生成

本稿では, Orange4 と同様のデータ構造によってプログラム中の全ての変数と式の値を把握することにより, 従来手法では行えなかった変異を実現する.

本稿で提案するプログラムの変異は「全ての変数の更新において更新値が元プログラムと等しい」という意味でのプログラムの等価変換に基づく. このような等価変換を「変数更新等価変換」と呼ぶことにする. 変数更新等価変換は必ず EMI でも等価な変換となる. ただし, Orion, Athena, Hermes が実装している変換は全て変数更新等価変換であり, その意味では本手法と同じである. 本手法では, プログラム中の全ての変数と式の値を把握することによって, 従来法では生成できなかった等価ミュータントを生成する.

```
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:
06: static volatile unsigned short x86 = 3U;
07: const signed short x88 = 386;
08: static volatile unsigned short t0 = 2428U;
09: volatile unsigned long long t1 = 2LLU;
10: static volatile signed char t3 = 9;
11: volatile signed char x92 = 115;
12:
13: int main (void)
14: {
15:   static const unsigned char x33 = 1U;
16:   static const volatile signed long long x41 = -1LL;
17:   volatile unsigned char x52 = 5U;
18:   const volatile unsigned char x65 = 38U;
19:   static signed long long x77 = 0LL;
20:   volatile signed int x87 = 511;
21:   const volatile signed long x89 = 2029215198009L;
22:   const volatile unsigned int x90 = 140U;
23:   volatile unsigned short t2 = 6570U;
24:   const unsigned long long x91 = 1255744988028LLU;
25:   static volatile unsigned long t4 = 14041369651127 82LU;
26:   static volatile signed char t5 = 17;
27:
28:   t0 = ((unsigned short)(((signed int)x87)&((signed short)x88)));
29:   t1 = ((unsigned long)(((signed long)x89)*((signed long long)x33)));
30:   t2 = ((unsigned short)(((unsigned int)x90)<<(((signed char)x77))));
31:   if(((signed char)(((unsigned short)x86)*((signed int)x41)))) {
32:     t3 = ((signed char)(((unsigned long)x77)<<((signed long)x52)));
33:     t4 = ((unsigned long)(((signed char)t3)|((unsigned long)x91)));
34:   }
35:   t5 = ((signed char)(((signed char)x92)%((signed char)x65)));
36:
37:   if (t0 == 386U) { OK(); } else { NG("t0", "%hu", t0); }
38:   if (t1 == 20292151980 09LLU) { OK(); } else { NG("t1", "%llu", t1); }
39:   if (t2 == 140U) { OK(); } else { NG("t2", "%hu", t2); }
40:   if (t3 == 0) { OK(); } else { NG("t3", "%hhd", t3); }
41:   if (t4 == 1255744988028LU) { OK(); } else { NG("t4", "%lu", t4); }
42:   if (t5 == 1) { OK(); } else { NG("t5", "%hhd", t5); }
43:
44:   return 0;
45: }
```

図 3　Orange4 が生成するプログラムの例

3.2　変 異 操 作

本稿で新たに導入する変異は, 1) データ型の変異, 2) 修飾子・記憶クラス指定子の変異, 3) 変数・配列・構造体・共用体の変異, 4) 制御文によるブロック化, の 4 つである.

3.2.1　データ型の変異

元プログラム中の変数の型を変更する. この際, 変数の値を考慮することによって未定義動作を含むプログラム生成を回避する.

図 4 にデータ型の変異の例を示す. 1 行目では x1 の型を long から int に, 2 行目では x2 の型を int から short に, 4 行目では m0 の型を int から unsigend char に, それぞれ変異させている.

変数の場合は変数の値を, 配列と構造体・共用体の場合は全ての要素とメンバ変数の値を収容できるデータ型の中から, ランダムに一つ型を選択して変異を行う. 図 4 の元プログラムの 2 行目の変数 x2 の値が 50000 なので, short 型に変異させると符号付き整数のオーバーフローが生じ, 未定義動作が生じるので, int より大きい型を選択する.

サイズの小さい型への変異は, 符号付き整数の場合には未定義動作を, 符号無し整数の場合にはラップラウンドによる値の変化を引き起こすため, 変数の値を把握して初めて可能になる.

(1) 元プログラム	(2) 変異後プログラム
1: long x1 = 50000;	1: int x1 = 50000;
2: int x2[2] = {1, 25000};	2: short x2[2] = {1, 25000};
3: struct s0 {	3: struct s0 {
4: int m0;	4: unsigned char m0;
5: };	5: };
6: s0 x3 = {100}	6: s0 x3 = {100}

図 4　データ型の変異

3.2.2 変数の修飾子・記憶クラス指定子の変異

元プログラム中の変数に対して 記憶クラス指定子 (static), 修飾子 (volatile, const) の挿入・削除をランダムに行う.

図 5 に例を示す. 1 行目 x0 では x1 の修飾子を volatile から const に, 2 行目 x1 では static を削除, 3 行目 x2 では static volatile を挿入している.

const の挿入は一度しか代入が行われない変数に対してのみ適用可能で, 変数利用の情報を利用してはじめて可能になる.

(1) 元プログラム	(2) 変異後プログラム
1: volatile int x0 = 1; 2: static int x1 = 2; 3: int x2 = 0;	1: const int x0 = 1; 2: int x1 = 2; 3: static volatile int x2 = 0;

図 5 修飾子・記憶クラス指定子の変異

3.2.3 変数・配列・構造体・共用体の変異

プログラム中の変数・配列・構造体をランダムに相互置き換えする.

変数・配列・構造体・共用体の変異の例を図 6 に示す. 元プログラム 1 行目の x0 は変数から構造体変数に, 2 行目の変数 x1 は配列に, 3 行目の配列 x2 は変数に, それぞれ変更させている.

変数を配列の要素にする場合は, まず配列の次元数と要素数をランダムに選択しその変数と同じデータ型の配列を生成し, 配列のどの要素を使用するかを選択し, それを変数と置き換える. 変数や配列を構造体・共用体化する場合は, 変異する変数, 配列と同じデータ型の変数, 配列をメンバとして持つ構造体を作成し, 作成した構造体・共用体を型とすることによって実現する. 配列・構造体変数・共用体変数を変数にする場合は全ての要素, メンバ変数をそれぞれ変数に置き換える.

変異を行うためには, 変異元の変数や配列などのデータ型の情報が必要となるため, 従来手法では行われていなかった.

(1) 元プログラム	(2) 変異後プログラム
	1: struct s0 { 2: int m0; 3: };
1: int x0 = 0; 2: int x1 = 1; 3: int x2[1] = 2 ; 4: int t0 = x0 + x1;	4: s0 x0 = { 0 }; 5: int x1[1] = { 1 }; 6: int x2 = 2; 7: int t0 = x0.m0 + x1[0];

図 6 変数・配列・構造体・共用体の変異

3.2.4 制御文による文のブロック化

プログラム中の式を条件式として利用することにより, 制御文による文のブロック化を行う.

制御文によるブロック化の例を図 7 に示す. 変異後プログラムの 9–11 行目では, 元プログラム 9 行目の代入文 `t1 = x2 * x1;` が 8 行目の式 `x0 * x1` を条件式とする if 文によってブロック化されている. 変異後プログラムの 13–18 行目では, 元プログラム 11 行目の代入文 `t3 = t0 | x0;` が 10 行目の式 `t1 / x1` を条件式とする while 文によってブロック化されている.

制御文のブロック化では if 文, for 文, while 文, switch 文を使用する. まず, ブロック化するプログラム中の文を選択する. 次に, 選択した文以前に実行される式の中から期待値が 0 以外の式をランダムに選択し, それを条件式として選択した文を制御文でブロック化する. 選択した文以前に実行される式の中から一つ式を選択する. ループ文の場合は図 7 の変異後プログラムの 16 行目のように break 文を追加し無限ループを回避する. ループ文のブロック内に直に break 文があると, コンパイラは ループ回数が 1 回であると判定して, 最適化によってループ文を削除してしまうため, break 文は if によりブロック化する.

(1) 元プログラム	(2) 変異後プログラム

1: int main (void){
2: int x0 = 1;
3: int x1 = 1;
4: int t0 = 1;
5: int t1 = 1;
6: int t2 = 1;
7: int t3 = 1;
8: t0 = x0 * x1;
9: t1 = x2 + x0;
10: t2 = t1 / x1;
11: t3 = t0 | x0;
12: return 0;
13: }
	1: int main (void){
	2: int x0 = 1;
	3: int x1 = 1;
	4: int t0 = 1;
	5: int t1 = 1;
	6: int t2 = 1;
	7: int t3 = 1;
	8: t0 = x0 * x1;
	9: if(x0 * x1) {
	10: t1 = x2 + x0;
	11: }
	12: t2 = t1 / x1;
	13: while(t1 / x1) {
	14: t3 = t0 | x0;
	15: if(x2 * x0) {
	16: break;
	17: }
	18: }
	19: return 0;
	20: }

図 7 制御文による文のブロック化

4. 実装と実験結果

4.1 実　　装

提案手法に基づくコンパイラのテストシステムを Perl 5.20.2 で実装した. 本システムは Ubuntu Linux 等で動作する.

実験では, Orange4 で検出した過去のコンパイラのエラープログラムを元プログラムとして, Orange4 が保持する全ての変数の型や値などの意味情報を用いて変異操作を行った.

図 3 のプログラムを元に生成したミュータントの一例を図 8 に示す. 14–19 行目と 24–35 行目では修飾子記憶およびクラス指定子の変異, データ型の変異を行っている. 6–12, 15, 20, 26–27 行目ではデータ型の変異を行っている. 制御文によるブロック化によって 41 行目では　39 行目の代入文の右辺式を条件式として if 文によるブロック化, 46 行目では 37 行目の代入文の右辺式を条件式として while 文によるブロック化を行っている.

4.2 実　　験

4.2.1 GCC-5.5.0 のテスト

Orange4 が GCC-4.4 で検出したエラープログラムのうち GCC-5.5.0 ではそのエラーを引き起こす不具合が修正されているプログラム 28 件を元プログラムとして, GCC-5.5.0 をテストした. テスト結果を表 1 に示す. “#time [h]” はテストの実行時間, “#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-5.5.0 で検出したエラー 159 件のうち, 重複したエラーを除外するとエラーは 2 件であった. GCC-5.5.0 で検出したエラーとその元プログラムのエラー箇所を残して不要部分を削除し最小化したものを 図 9, 図 10 に示す.

表 1 実験結果 1

コンパイラ	#time [h]	#test	#error
GCC-5.5.0	100	377,415	159

(CPU Intel(R) Xeon(R) E3–1276 v3 3.60GHz, RAM 16GB)

図 9 (a) は変異元プログラム, 図 9 (b) は GCC-5.5.0 で検出したエラープログラムをそれぞれ最小化したものである. 変数 d の期待値は 8 だが, -O3 オプションでコンパイル・実行すると `__builtin_abort();` が実行されてエラーとなる. 元プログラム (a) の不具合は GCC-5.5.0 では修正されているが, 4 行目で変数 b に const 修飾子を付加した (b) のプログラムは GCC-5.5.0 でエラーを引き起こす.

```
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:
06: struct s0 {
07:   volatile signed long m0;
08: };
09:
10: union u0 {
11:   const signed long long m0;
12: };
13:
14: const volatile signed long long x85 = -270876425514473LL;
15: static const signed short x86[3] = { 0, 0, 3 };
16: static const volatile signed long long x88 = 386LL;
17: static volatile signed short t0 = 0;
18: signed long t1 = 0L;
19: static signed short t3 = 0;
20: struct s0 x92 = { 115L };
21:
22: int main (void)
23: {
24:   const volatile unsigned char x33 = 1U;
25:   static volatile signed long long x41 = -1LL;
26:   union u0 x52 = { 5LL };
27:   volatile unsigned short x65[1] = { 38U };
28:   unsigned short x77 = 0U;
29:   const volatile signed long long x87 = 511LL;
30:   static const volatile signed long long x89 = 2029215198009LL;
31:   static const volatile signed short x90 = 140;
32:   static unsigned int t2 = 0U;
33:   static const volatile signed long long x91 = 1255744988028LL;
34:   unsigned long long t4 = 0LLU;
35:   volatile signed long t5 = 0L;
36:
37:   t0 = ((unsigned short)(((signed int)x87)&((signed short)x88)));
38:   t1 = ((unsigned long)(((signed long)x89)*((signed long long)x33)));
39:   t2 = ((unsigned short)(((unsigned int)x90)<<(((signed char)x77))));
40:   if(((signed char)(((unsigned short)x86[2])*((signed int)x41)))) {
41:     if((unsigned short)(((unsigned int)x90)<<(((signed char)x77)))) {
42:       t3 = ((signed char)(((unsigned long)x77)<<((signed long)x52.m0)
          ));
43:       t4 = ((unsigned long)(((signed char)t3)|((unsigned long)x91)));
44:     }
45:   }
46:   while(((unsigned short)(((signed int)x87)&((signed short)x88)))) {
47:     t5 = ((signed char)(((signed char)x92.m0)%((signed char)x65[0])));
48:     if(((unsigned long)(((signed char)t3)|((unsigned long)x91)))) {
49:       break;
50:     }
51:   }
52:
53:   if (t0 == 386) { OK(); } else { NG("t0", "%hd", t0); }
54:   if (t1 == 2029215198009L) { OK(); } else { NG("t1", "%ld", t1); }
55:   if (t2 == 140U) { OK(); } else { NG("t2", "%u", t2); }
56:   if (t3 == 0) { OK(); } else { NG("t3", "%hd", t3); }
57:   if (t4 == 1255744988028LLU) { OK(); } else { NG("t4", "%llu", t4); }
58:   if (t5 == 1L) { OK(); } else { NG("t5", "%ld", t5); }
59:
60:   return 0;
61: }
```

図 8　図 2 から生成した等価ミュータント

```
01: int main (void)
02: {
03:   int a = 0x2000;
04:   int b = 1;
05:   long c = 0x1000000000000000L;
06:   int d = (-0x4000000000000L*a*b) / (c/-1);
07:   if (d != 8) __builtin_abort();
08:   return 0;
09: }
```

(a) 元プログラム

```
01: int main (void)
02: {
03:   int a = 0x2000;
04:   const int b = 1;
05:   long c = 0x1000000000000000L;
06:   int d = (-0x4000000000000L*a*b) / (c/-1);
07:   if (d != 8) __builtin_abort();
08:   return 0;
09: }
```

(b) 検出したエラープログラム

図 9　GCC-5.5.0 で検出したエラープログラム (1)

図 10 (a) は変異元プログラム, 図 10 (b) は GCC-5.5.0 で検出したエラープログラムをそれぞれ最小化したものである. 元プログラム (a) の不具合は GCC-5.5.0 では修正されているが, 4 行目の変数 b の修飾子を const から volatile に変更し, 5 行目変数 c に const 修飾子が付加した (b) のプログラムは GCC-5.5.0 でエラーを引き起こす.

```
01: int main (void)
02: {
03:   volatile int a = -1;
04:   const int b = 2;
05:   long c = -0x4000000000000000L;
06:   long d = (0x7FFFFFFFFFFFFFFFL/a) - (c*(2&b));
07:   if (d != 1) __builtin_abort();
08:   return 0;
09: }
```

(a) 元プログラム

```
01: int main (void)
02: {
03:   volatile int a = -1;
04:   volatile int b = 2;
05:   const long c = -0x4000000000000000L;
06:   long d = (0x7FFFFFFFFFFFFFFFL/a) - (c*(2&b));
07:   if (d != 1) __builtin_abort();
08:   return 0;
09: }
```

(b) 検出したエラープログラム

図 10　GCC-5.5.0 で検出したエラープログラム (2)

4.2.2　LLVM/Clang-3.5.2 のテスト

Orange4 が LLVM/Clang-3.0 で検出したエラープログラムでかつ LLVM/Clang-3.5.2 ではそのエラーを引き起こす不具合が修正されているプログラム 128 件を元プログラムとして, LLVM/Clang-3.5.2 をテストした. 結果を表 2 に示す.

表 2　実験結果 2

コンパイラ	#time[h]	#test	#error
LLVM/Clang-3.5.2	100	399,309	8

(CPU Intel(R) Xeon(R) E3–1276 v3 3.60GHz, RAM 16GB)

LLVM/Clang-3.5.2 で検出したエラー 8 件のうち重複したエラーを除外すると 1 件であった. LLVM/Clang-3.5.2 で検出したエラーとその元プログラムを図 11 に示す.

図 11 (a) は変異元プログラム, 図 11 (b) は Clang-3.5.2 で検出したエラープログラムをそれぞれ最小化したものである. 元プログラム (a) は LLVM/Clang-3.0 で -O3 オプションでコンパイルするとコンパイラがクラッシュするが, LLVM/Clang-3.5.2 ではこの不具合が修正されている. (b) では変数 b, c, d, f の型や修飾子が変化しているが, これを LLVM/Clang-3.5.2 で -O3 オプションでコンパイルするとコンパイラがクラッシュする.

4.3　考　　察

今回検出したエラーは, 全て修飾子・記憶クラス指定子やデータ型の変異によって検出されたものであった. 変異元として利用したエラープログラムの多くは, コンパイラの最適化処理の一つである式を計算した結果の定数値に置き換える定数伝搬という処理の不具合に起因するものであった. 修飾子・記憶クラス指定子やデータ型の変異はこのような不具合の検出に有効であると考えられる.

本実験で検出したエラーは全て修飾子やデータ型の些細な変異によって発生していた. 今回は Orange4 で検出したエラープログラムに最小化を行わないものを元プログラムとして利用したが, 実験でエラーを検出したプログラムは最小化後のプログラムを変異させたものになっていたため, 最小化後のエラープログラムを元プログラムとして変異を行うことも考えられる.

本実装では, 元プログラム中の全ての変数の型と値の更新の情報を得るために, Orange4 の意味情報を利用するため, Orange4 以外のテストシステムで生成されたプログラムを元プログラム

```
01: volatile int a = -1;
02: long b = 1L;
03: volatile long c = 1L;
04: long d = -0x7FFFFFFFFFFFFFFFL;
05: int main (void)
06: {
07:   volatile int e = 1;
08:   long f = -0x3FFFFFFFFFFFFFFFL;
09:   short g = (-1/(-(short)(-1/b)))*2;
10:   int h = ((c+f+(short)(g/e))+(d%(a+f)))%
              ((0x7FFFFFFFFFFFFFFFL*(g<0))/a);
11:   if (h != 0) __builtin_abort();
12:   return 0;
13: }
```

(a) 元プログラム

```
01: const int a = -1;
02: int b = 1;
03: static int c = 1;
04: const long d = -0x7FFFFFFFFFFFFFFFL;
05: int main (void)
06: {
07:   volatile int e = 1;
08:   volatile long f = -0x3FFFFFFFFFFFFFFFL;
09:   short g = (-1/(-(short)(-1/b)))*2;
10:   int h = ((c+f+(short)(g/e))+(d%(a+f)))%
              ((0x7FFFFFFFFFFFFFFFL*(g<0))/a);
11:   if (h != 0) __builtin_abort();
12:   return 0;
13: }
```

(b) 検出したエラープログラム

図 11　LLVM/Clang-3.5.2 で検出したエラープログラム

として使用することができない. 一般的なプログラムに対して変異操作を適用することは今後の課題である.

5. む　す　び

本稿では, 等価ミュータント生成により C コンパイラのテストのバリエーションを増強する手法を提案した. 実験の結果, GCC-5.5.0, LLVM/Clang-3.5.2 において過去のコンパイラのエラープログラムを元プログラムとすることによって不具合を検出できた.

今後の課題としては, キャスト演算の型変異, 変数の持つ値の変異など変異操作の強化, アセンブリ比較や実行時間差比較による最適化性能テストすることが挙げられる.

謝　　辞

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] 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).

[2] 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).

[3] 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).

[4] S. Takakura, M. Iwatsuji, and N. Ishiura: “Extending Equivalence Transformation Based Program Generator for Random Testing of C Compilers,” in *Proc. A-TEST 2018*, pp. 9–15 (Nov 2018).

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

[6] Vu Le, Mehrdad Afshari, Zhendong Su: “Compiler validation via equivalence modulo inputs,” in *Proc. PLDI 2014*, pp. 216–226 (June 2014).

[7] 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).

[8] Chengnian Sun, Vu Le, Zhendong Su: “Finding Compiler Bugs via Live Code Mutation,” in *Proc. OOPSLA 2016*, pp. 849–863 (Oct 2016).

[9] Qirun Zhang, Chengnian Sun, P. Cuoq, Zhendong Su: “Skeletal Program Enumeration for Rigorous Compiler Testing,” in *Proc. PLDI 2017*, pp. 347–361 (June 2017).