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
VLD2015-112(2016-02)

等価変換に基づくテストプログラム生成による C コンパイラのランダムテスト

中村　和博†　　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 コンパイラの信頼性向上を目的としたランダムテストにおいて, プログラムの等価変換によってテストプログラムを生成する手法を提案する. 従来のランダムプログラム生成法では, 未定義動作回避のために, 生成できるプログラムの構文に制約が生じるという課題があった. プログラムの等価変換に基づくテストプログラム生成はこれを解決する一手法と考えられるが, これまでに提案されている手法は, 元となる正当なテストプログラムが必要であること, およびプログラム中の到達不能コードに対してしか変換を適用できないという課題があった. 本稿の手法では, 自明なプログラムから始めて等価変換の適用を繰り返すことによって新たなテストプログラムを生成することにより, これらの課題を解決する. また, 本手法では, 算術式中のオペランドの値の分布を制御することによってエラー検出率の向上を図ることも可能である. 本手法に基づくランダムテストシステム Orange4 を実装し, 実験を行ったところ, GCC-6.0.0 および LLVM/Clang-3.9 (それぞれ, 2015 年 11 月, および 2016 年 2 月時点における最新バージョンの開発版) において, 不具合を検出することができた. また, GCC-4.5.0 において, 従来手法では検出できない不具合を検出することができた.

キーワード　コンパイラ, 信頼性, ランダムテスト, 等価変換, 最小化, Orange4

Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation

Kazuhiro NAKAMURA† and Nagisa ISHIURA†

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

Abstract　This article proposes a method of generating test programs for random testing of C compilers based on equivalence transformation on C programs. Although equivalence transformation on programs is a promising way of deriving new test programs without undefined behavior, existing test generation methods had shortcomings that they needed valid seed test programs and that they applied only addition/deletion of statements to/from unreachable portion of the seed test programs. Our method is based on program transformation on live codes, and can generate a vast variety of test programs starting from a trivial C program. Furthermore, our method can control the resulting operand values of the subexpressions in generated test programs, which contributes to a higher bug detection capability. A random test systems, Orange4, implemented based on the proposed method has detected bugs in the latest development versions of GCC-6.0.0 and LLVM/Clang-3.9. It has also detected a bug in GCC-4.5.0 which cannot be detect by our previous methods.

Key words　compiler, reliability, random testing, equivalence transformation, minimization, Orange4

1. はじめに

コンパイラはソフトウェア開発の基盤となるツールである. コンパイラの不具合は, 開発対象のシステムの信頼性に大きな影響を及ぼすため, コンパイラには非常に高い信頼性が要求される. 成熟したコンパイラにおいては, 構文解析部や基本的なコード生成部で不具合が発生することはほとんどない. しかし, コンパイラの最適化処理には性能向上のため日々改良が加え続けられているため, ここに不具合が発生する余地がある. このため, コンパイラの信頼性を確保するためのテストは重要な課題となる.

コンパイラの開発では, その段階に応じて, 開発者が用意した

テストプログラムやテストスイートを用いて徹底的なテストが行われる．テストスイート [1]〜[4] はコンパイラをテストするための膨大な数のテストプログラム群であるが, テストスイートによるテストを経てもなお不具合が潜在する場合がある．そのような不具合を検出するための手法として, ランダムに生成したプログラムを用いてコンパイラをテストするランダムテストが用いられる [5].

Csmith [6] は広範な構文のテストを行うことができるランダムテストで, GCC-4.5 や LLVM/Clang-2.8 等において, 3 年間で約 325 個の不具合を検出している．Csmith は 1 つのプログラムを複数のコンパイラでコンパイルしてエラー判定を行う differential testing [7] に基づいているが, ゼロ除算などの未定義動作をプログラムの静的解析だけに基づいて回避しなければならないため, 生成できる文や式に制約が生じるという課題があった.

これに対し Orange3 [8] は, テストプログラムに期待される実行結果をプログラムの生成と同時に求めることによって, より正確に未定義動作の回避を行う手法を提案しており, x86 用の GCC-5.0.0 や LLVM/Clang-3.6 等において, Csmith では検出できなかった不具合を検出している．しかし, 構文が複雑になると未定義回避の処理が困難になるため, 主として生成できるテストプログラムは算術式と代入文のみを含むものであり, その拡張も for 文を対象としたもの [9] に限られていた.

Orange3 において, 未定義動作を含まないランダムな算術式を生成する新たな方法として, 中橋・由良らは算術式の解析木の導出による式生成手法を提案している [10], [11]．これは, 式の期待値を先に決定し, 演算の結果がその値になるように算術式を生成する手法である．この手法では, 未定義動作を含まない状態を維持しながらテストプログラムを複雑化することが容易に行えるという利点がある.

文献 [10], [11] のような手法は, 実行結果が変わらないようにプログラムを書き換えるという意味において, プログラムの等価変換に基づいていると考えられる．プログラムの等価変換に基づくランダムテストは, Proteus [12], Athena [13] 等がここ 3 年で大きな成果を挙げている手法である．Athena は Csmith 等を用いて用意したテストプログラムに対し, 実行されない部分にコードの挿入/削除を行うことにより等価なプログラムを生成して, テストを行う手法で, GCC や LLVM/Clang 等のコンパイラにおいて, 19 ヶ月で 72 個の不具合を検出している．この手法は, 既存のテストプログラムから大量のテストプログラムを作れるというメリットがある．しかし, 元となる正しいテストプログラムを準備する必要があること, およびプログラム中の到達不能コードに対してしか変換操作を適用していないという課題がある.

そこで本稿では, 等価変換に基づいて新たなテストプログラムを生成する手法を提案する．本手法では自明なプログラムを用意し, それに対して文の追加や式の導出等の変換ルールをランダムに適用することによって, 複雑なテストプログラムを生成する．未定義動作が生じないように変換ルールを定義すれば, プログラム生成後に未定義動作の回避が不要になるため, Orange3 では難しかった機能拡張が行い易くなる．また, 文献 [10], [11] の手法を導入することにより, 式の期待値とオペランドの値を制御して, 不具合検出能力の向上を図ることもできる.

本手法に基づくランダムテストシステムを実装した結果, GCC-6.0.0 および LLVM/Clang-3.9 (それぞれ, 2015 年 11 月, および 2016 年 2 月時点における最新バージョンの開発版) において, 不具合を検出することができた．また, GCC-4.5.0 において, Orange3 では検出できない不具合を検出できた.

以下, 2 章ではコンパイラのランダムテストとそのアプローチおよび Orange3 について述べた後, 3 章にて提案する等価変換に基づくテストプラグラム生成の手法を述べる．4 章では実装と評価を示し, 5 章で結論を述べる.

2. コンパイラのランダムテスト

2.1 コンパイラのランダムテストとそのアプローチ

コンパイラのランダムテストは, ランダムに生成したテストプログラムをコンパイル・実行し, その結果が正しいかどうかを判定するという処理を, 設定した時間もしくは回数だけ行うものである.

コンパイラのランダムテストでは, テストプログラムの正誤判定と未定義動作の回避をどのように行うかが課題となる．コンパイラが正しいかどうか判定するためには, ランダムに生成したプログラムの正しい実行結果がわかっていなくてはならない．未定義動作とは, ゼロ除算, 符号付き整数のオーバフロー, 初期化されていない変数の参照等であり, その動作は言語仕様 [14] で「何ら要求を課さない」と定められている．未定義動作を含むプログラムは任意の計算結果が正しいことになるため, テストプログラムとして生成することは避けなければならない.

コンパイラのランダムテストには, 大きく分けて,
・テストプログラムを 2 つの異なるコンパイラを用いてコンパイル・実行し, その結果を比較する差分テスト [7] (図 1(a)),
・テストプログラムの実行結果をあらかじめ計算し, コンパイル・実行した結果と比較する方法 (図 1(b)),
・テストプログラムに等価変換を施して別のプログラムを得, 二つのプログラムの実行結果を比較する方法 (図 1(c)),
が存在する．等価変換に基づく手法では, 何らかの方法で元となる正当なテストプログラムを生成しなければならない．また, これまでに提案されている手法の等価変換は, 元のプログラムの実行に影響を与えない文の追加/削除に限られていた.

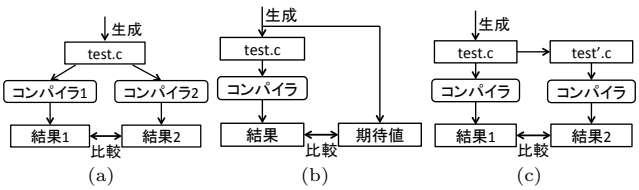

図 1　コンパイラのランダムテストのアプローチ

2.2 算術最適化を対象としたランダムテスト Orange3

Orange3 [8] は, 算術最適化を対象としたランダムテストである．図 1(b) の期待値計算に基づく手法であり, プログラムの生成と同時に計算する期待値に基づいて未定義動作の回避を行う.

Orange3 が生成するテストプログラムの例を図 2 に示す．5〜29 行目は変数の宣言であり, 31〜35 行目の算術式の計算結果が期待値と等しいかを, 37〜41 行目でチェックしている．変数名が x から始まる変数は, 式の右辺のみで使われるもので, t から始まる変数は式の計算結果 (期待値) を代入するためのも

のである．この例は 1 プログラム中に含まれる演算子の数と式の数の積を 20 と設定し生成したものだが, 実際にはコンパイラの完成度に合わせて, 50〜3000 程度に設定する．

```
01: #include <stdio.h>
02: #define OK()  printf("@OK@\n")
03: #define NG(fmt,val)  printf("@NG@ (test = " fmt ")\n",val)
04:
05: static const volatile float x0 = -9.0F;
    ...
15: signed int t3 = -93467547;
16:
17: int main (void)
18: {
19:     static const double x3 = -7461802.0;
        ...
29:     signed long t4 = 285128L;
30:
31:     t0 = (x10+((x16<(x9/x13))+x4));
32:     t1 = ((x6*(x17<<((((signed int)x12)+k22)>>x21)))&&x19);
33:     t2 = ((t0/(x12<=(x12>=x17)))&((signed int)x12));
34:     t3 = (((((signed int)x3)+k25)>>(x15+x13))>(x2&((signed int)x0)));
35:     t4 = (x10|(x2-((x19==x12)>>x27)));
36:
37:     if (t0 == 2284795846113651LL) { OK(); } else { NG("%lld", t0); }
        ...
41:     if (t4 == -1L) { OK(); } else { NG("%ld", t4); }
42:
43:     return 0;
44: }
```

図 2　Orange3 のテストプログラムの例

Orange3 [8] は算術式のテストしか行えないが, 不具合検出能力の向上を目的としてループ文の追加 [9] 等の拡張が行われている．Orange3 では文や式の生成後にプログラムを解析して未定義動作の回避処理を行うが, 構文が複雑になるとその処理が困難になるため, これ以上の拡張は難しいという課題がある．

また, ランダムに式と値を生成するランダムテストの場合には, 算術式中のオペランドの値が偏るという課題もある．表 1 は, Orange3 で生成したテストプログラムにおける算術式の値の出現頻度を型ごとに集計したものである (式数と演算子数の積を 1000 とし, テストプログラムを 100 本生成した場合の結果である)．“型” は算術式の型, MIN は各型の最小値, MAX は最大値である．表より, 0 が他の値に比べて多く出現していることがわかる．この偏りは式中に出現する比較演算の結果が 0 か 1 であることに起因すると考えられる．この集計は各式の期待値に関するものであるが, 式中のオペランドの値にも同様の偏りが発生していると考えられる．コンパイラには特定のオペランド値に対して適用される最適化があるため, 値の偏りが存在すると, テストが特定の最適化に集中してしまう可能性がある．

表 1　Orange3 における算術式の値の出現頻度

型	MIN〜−1	0	1	2〜MAX
signed int	8.9%	42.1%	23.9%	25.1%
unsigned int		31.7%	3.3%	65.0%
signed long	10.9%	16.2%	1.5%	71.4%
unsigned long		27.0%	2.3%	70.7%
signed long long	11.9%	15.9%	1.1%	71.1%
unsigned long long		29.9%	3.5%	66.6%

2.3　算術式の解析木の展開に基づくランダムテスト

中橋・由良らは解析木の導出の繰り返しによって, 未定義動作を含まない算術式を生成する手法を提案している [10], [11]．この手法では, はじめに式の期待値の型と値をランダムに決定し, 次に, 使用する演算子およびオペランドの型と値を決定する．この際, オペランドの値は未定義動作が発生しないように決定する．オペランドには新しく宣言した変数を使用してもよいし, 宣言済みの変数を再利用してもよい．

式生成の例を図 3 に, 実際のプログラムを図 4 に示す．はじめに, ランダムに決定した期待値が int 型の 36 であるとする (図 4(a))．これに対して, 使用する演算子が << と決定されたとすると, 左オペランドにどんな値が来ても未定義動作が発生しないように, 最小値 0, 最大値 2 の範囲から選択する (負数によるシフトは未定義であり, 3 以上の数によるシフトで 36 は生成できないため)．右オペランドを int 型の 2 に決定したとすると, 左オペランドの値は 9 となる (図 4(b))．次に, 左オペランド (int 型の 9) を展開する．使用する演算子が * で, 右オペランドを short 型の 3 に決定したとすると, 左オペランドの値は 3 となる (図 4(c))．

この手法を用いれば, 未定義動作が起きないように式を生成することができるため, 式の生成後に未定義回避のための修正を行う必要がなくなる．さらに, オペランドは算術式を導出する過程で生成されるため, 未定義動作が発生しない範囲内であれば値を自由に制御できるという利点もある．文献 [11] では, 期待値の分布を変更することにより, ランダムテストのエラー検出率が変化することが報告されている．

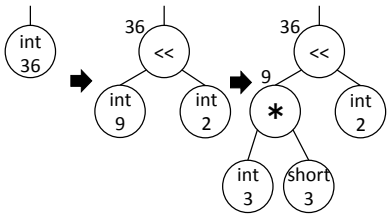

図 3　値の展開による式生成の例

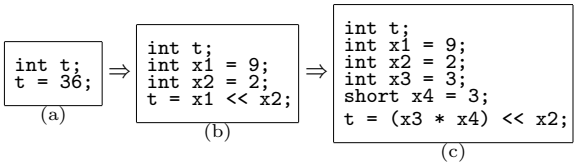

図 4　値の展開による式生成の例 (プログラム)

3.　等価変換に基づくテストプログラム生成によるランダムテスト

3.1　等価変換に基づくテストプログラム生成

本章では, C コンパイラのランダムテストにおいて, 等価変換を繰り返し適用することによりテストプログラムを生成する手法を提案する．本手法により, Athena [13] において, 等価変換を行う際に何らかの正当なテストプログラムを用意する必要があるという課題, およびプログラム中の未実行部分にしか等価変換を行えないという課題の解決を図る．また, 本手法では等価変換の一つに文献 [10], [11] の算術式の解析木の導出を用いることにより, Orange3 [8] の機能追加の拡張が難しいという課題を解決することができる．さらに, 式の期待値とオペランド値を制御することにより, 不具合検出能力の向上を図る．

本手法のテストプログラム生成は以下の流れで行う．

(1) 自明なプログラムを用意する．

(2) 変換ルールを適用する回数をランダムに決定する．

(3) プログラムに対して適用する変換ルールをリストからランダムに選択する．

(4) プログラムに対して変換ルールを適用し, (3) に戻る．この際, (2) で決定した回数に達すれば処理を終了し, テストプログラムを出力する．

(1) で用意する自明なプログラムの例を図 5 に示す．このプログラムは, ヘッダのインクルードとエラー出力用のマクロ,

return 0; のみを記述した main 関数から成る.

本手法により生成されるテストプログラムの例を図 6 に示す. 5〜69 行目で変数の宣言を行なっている. 71〜112 行目は for 文, if 文, 代入文である. 114〜128 行目では各代入文の計算結果が期待値と等しいかを比較している.

```
1: #include <stdio.h>
2: #define NG() printf("@NG@\n")
3:
4: int main (void) {
5:     return 0;
6: }
```

図 5　自明なプログラムの例

```
001: #include <stdio.h>
002: #define OK()  printf("@OK@\n")
003: #define NG(test,fmt,val)  printf("@NG@ ("test" = " fmt ")\n",val)
004:
005: static signed long long x40 = 5473828800LL;
     ...
037: static unsigned short t18 = 9U;
038:
039: int main (void)
040: {
041:     volatile unsigned int x12 = 43794490U;
         ...
068:     signed long x103 = 143L;
069:     signed int i;
070:
071:     for( i = ((signed int)(x40/((signed long)x36)));
072:          i > ((signed int)(x41-x42));
073:          i -= ((signed int)(x43-((signed long)x42)))) {
074:         ...
085:         if( ((unsigned long)(((unsigned char)x68)/x69)) ) {
086:             t7 = ((unsigned long)(x70>>((signed int)x52)));
087:             t8 = ((unsigned char)(x71>>((unsigned long long)t2)));
088:         }
089:         else {
090:             t9 = ((unsigned long)(x72%((signed long long)x34)));
091:             t10 = ((signed char)(x75%((signed long long)x56)));
092:             t11 = (((signed char)x21)*x78);
093:             t12 = ((unsigned int)(x81&x82));
094:         }
             ...
112:     }
113:
114:     if (t1 == 3863U) { OK(); } else { NG("t1", "%hu", t1); }
         ...
128:     if (t17 == 2765) { OK(); } else { NG("t17", "%hd", t17); }
129:
130:     return 0;
131: }
```

図 6　本手法で生成するテストプログラムの例

3.2　変換ルール

本稿では, C 言語仕様に含まれる構文のうち, 算術式, if 文, for 文を対象とする. 本手法における等価変換とは, プログラムの変換の前後で実行したときの出力が変化しないものを指す. したがって, すべてのルールは適用前後でプログラムの出力が変化しないという条件を満たす. 等価変換を適用する際の変換ルールを以下に示す.

（1） 変数宣言の追加

変数宣言を一つ追加する. 変数の型は, char から long long の符号付き/符号なしの整数型である. 初期化する値の範囲は, 各変数の型の最小値から最大値で, 修飾子は “const”, “volatile”, “const volatile”, “なし”, の 4 種類からランダムに選択する.

変数宣言と同時に初期値をランダムに決定し初期化する. 変数宣言を追加する箇所は, グローバル変数は main 関数の直前で, ローカル変数は main 関数が始まってすぐの箇所である.

図 7 の (a) に, このルールを 4 回適用した例が (b) である.

（2） 定数代入文の追加

宣言済みの変数に対しての定数代入文を追加する. ただし, 同じ変数に対しての代入は 1 回のみとする.

図 8 の (a) に, このルールを 1 回適用した例が (b) である.

（3） if 文の追加

図 9 の流れに従って if 文を追加する. はじめに, then 節のみか, else 節も追加するのか決定する. then 節のみの場合は, プログラム中に含まれる文 (代入文, if 文, for 文) のリストを三つに分割し, それぞれ if 文の直前, 内部, 直後に記述することによって, if 文の追加を行う. 分割した文リスト i は空でもよいものとする. else 節も追加する場合は, 文リストを四つに分割し, それぞれ if 文の直前, then 節, else 節, else 節の直後に記述する. if 文の条件式の値は 0 かそれ以外の乱数とする.

図 10 の (a) に, if 文の追加を行った例が (b) である. (b) に, if 文の追加を行った例が (c) である.

（4） for 文の追加

図 11 の流れに従って for 文を追加する. for 文の追加は, 文リストを三つに分割し, それぞれ for 文の直前, 内部, 直後に記述することにより行う.

図 12 の (a) に, for 文の追加を行った例が (b) である.

なお本稿では, for 文のループ回数を 0 回か 1 回に限定する. 後述のルール (6) の適用でループ回数を決定する式に volatile 変数が出現すると, ループ回数はコンパイル時には決定できないため, この制約によりテストできるループ最適化が著しく限定されることはないと考えられる (注1).

（5） 変数の期待値照合の追加

変数の値が正しいかをチェックするために, 期待値照合を挿入する. 挿入する箇所は return 0; の直前である.

図 13 の (a) に, このルールを 1 回適用した例が (b) である.

（6） 定数 (代入文, if の条件文, for の引数) の式への展開

ルール (2)〜(4) で生成された代入文, if の条件文, for の引数に使われている定数を文献 [10], [11] の手法によって展開する.

図 14 の (a) に, このルールを 1 回適用した例が (b) である.

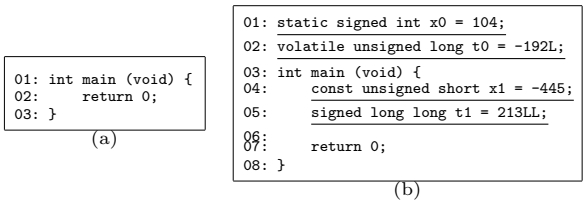

図 7　ルール (1) の適用例

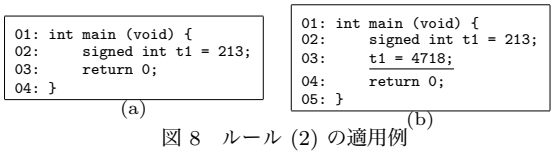

図 8　ルール (2) の適用例

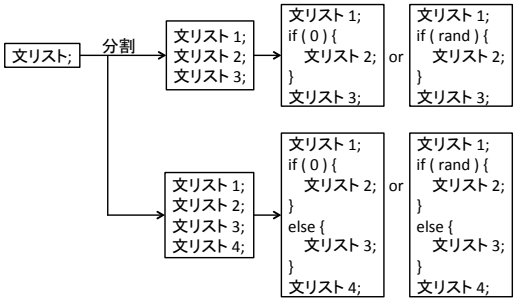

図 9　if 文の追加

3.3　式の期待値とオペランドの値の制御

本手法では, 式の期待値と式中のオペランドの値を制御する

(注1)：ただし, コンパイル時に決定できるループ回数に依存した最適化等はテストできない.

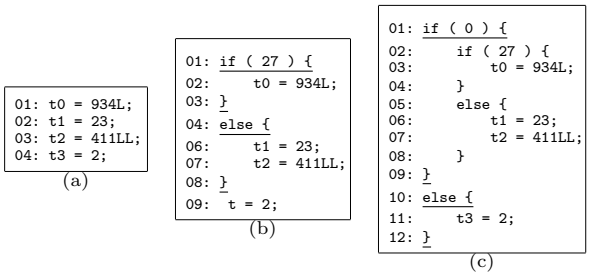

図 10　ルール (3) の適用例

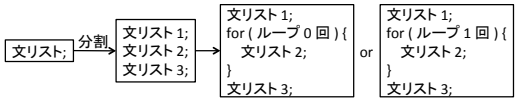

図 11　for 文の追加

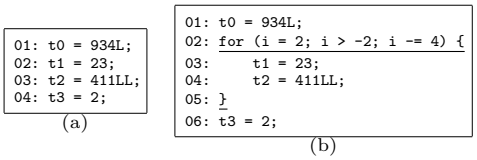

図 12　ルール (4) の適用例

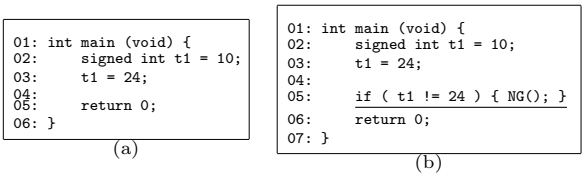

図 13　ルール (5) の適用例

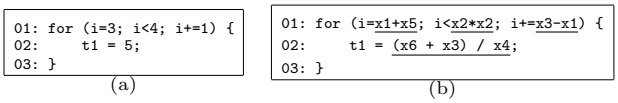

図 14　ルール (6) の適用例

ことが可能である. 期待値とオペランドの値の制御によって, 任意の値を出現しやすくすることができる. 期待値は, ルール (2) を適用する際に, 型の範囲内で任意の値に設定できる. オペランドの値は, ルール (5) を適用する際に制御を行う.

図 15 に値 r を算術式に展開する際の例を示す. はじめに, r という値に対して演算子を決定する (+). 次に左辺に使うことのできる値の範囲を調べる ($r-\mathrm{MAX}$〜$r-\mathrm{MIN}$). この時の MAX, MIN は変数の型の最大値, 最小値を表す. 範囲が決まると, その範囲内で乱数を用いて, 左辺の値を決定する.

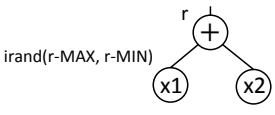

図 15　値の展開の例

本手法では, 左辺の乱数生成の際に値を範囲内で任意に設定できる. 例えば, $r-\mathrm{MAX}$, -1, $r-\mathrm{MIN}$ をそれぞれ 10%, その他の乱数を 70% と設定すると, $r-\mathrm{MAX}$, -1, $r-\mathrm{MIN}$ の値が出現し易くなる.

3.4　テストプログラムの最小化

コンパイラの不具合を検出した際, デバッグのため, 保存されたプログラムからエラーの原因となる箇所を特定する必要があるが, テストプログラムは多くの場合数千行〜数万行に及び, そのまま解析することは困難である. そこで不具合を検出したテストプログラムの最小化 (エラーを検出できる状態を維持しつつ, プログラムをできる限り縮約すること) が必要となる.

本手法における最小化は, (1) 二分探索による式の削減, (2) 式の最小化, (3) if 文 と for 文の木の最小化, (4) 変数, 定数の削減 の順で行う. これらの操作のいずれかで効果があった場合 (1) に戻り, どの操作も効果がなくなるまで一連の流れを繰り返す.

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 Perl 5.18.2 で実装した. 本システムは Windows, Mac OS X, Ubuntu Linux 等で動作する.

GCC, LLVM/Clang の 5 つのバージョンのコンパイラに対して, Orange3 [8] と本手法によるランダムテストを適用した. 実験時間はすべてのコンパイラで 72 時間である. 最適化オプションは `-O0`, `-O3`, `-Os` の 3 つを使用した. 一つのテストプログラム中に含まれる演算子数と式数の積は 1000 とした.

実験の結果を表 2 に示す. “Orange3” は Orangee3 でテストを行った結果である. “提案手法 (一様分布)” は算術式の展開時に, 左のオペランドの値をランダムに選択したものである. “提案手法 (変更分布)” は生成可能な値の範囲の最小値を 10%, 最大値を 10%, -1 を 10%, 範囲内の乱数を 70% の確率で生成するように, 値の分布を変更したものである. “#test” は生成したテストプログラムの数, “#error” はエラーを検出した数である.

“提案手法 (一様分布)” ではエラーを検出することができなかったが, “提案手法 (変更分布)” では, Orange3 より多くのエラーを検出することができた.

図 16 は GCC-6.0.0(注1) で検出したエラープログラムを最小化したものである. 変数 `t` の期待値は 0 なので 10 行目の if 文の本体は実行されず, 12 行目の `return 0;` に到達するはずだが, `-O2` オプションでコンパイル・実行すると `__builtin_abort();` が実行されてプログラムは異常終了する. この件は GCC の Bugzilla に報告しており (注2), 修正が行われた.

```
01: #define INT_MIN ( -2147483647 - 1 )
02:
03: int main (void)
04: {
05:     int  x0 = INT_MIN;
06:     long x1 = 0L;
07:     int  x2 = 0;
08:     int  t  = ( 0 || ( INT_MIN - (int) ( x0 - x1 ) ) );
09:
10:     if ( t != 0 ) { x2 = t; __builtin_abort(); }
11:
12:     return 0;
13: }
```

図 16　GCC-6.0.0 で検出したエラープログラム

図 17 は表 2 の実験とは別に, LLVM/Clang-3.9(注3) で検出したエラープログラムを最小化したものである. 変数 `f` の期待値は 1 なので 9 行目の `__builtin_abort();` は実行されないはずだが, LLVM/Clang-3.9 において `-O1` オプションでコンパイル・実行するとプログラムは異常終了する. この件は LLVM

(注1)：gcc version 6.0.0 20151112 (experimental)
(注2)：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68528

表 2　Orange3 との比較実験の結果

コンパイラ (ターゲット)	Orange3		提案手法 (一様分布)		提案手法 (変更分布)	
	#test	#error	#test	#error	#test	#error
GCC-4.8.4 (x86_64-pc-linux)	57,051	0	29,871	0	30,187	16
GCC-5.2.1 (x86_64-pc-linux)	49,402	0	30,208	0	26,659	16
GCC-6.0.0 (x86_64-pc-linux)	37,417	0	24,762	0	21,905	2
LLVM/Clang-3.6 (x86_64-pc-linux)	54,972	1	32,778	0	29,467	4
LLVM/Clang-3.8 (x86_64-pc-linux)	44,685	1	51,985	0	24,258	0

変更分布: MIN 10%, −1 10%, MAX 10%, rand 70%

time: 72 (h), size: 1000

CPU:Intel(R) Core(TM) i7–4930K 3.40GHz, RAM:15.6GiB

の Bugzilla に報告しており [(注4)], 修正が行われた.

```
01: int x =  0;
02: int y =  -1;
03:
04: int main (void)
05: {
06:     int a = 0x7fffffff + x;
07:     int b = 0x7fffffff + y;
08:     int f = (unsigned int) a >= (unsigned int) b;
09:     if ( f != 1 ) { __builtin_abort(); }
10:     return 0;
11: }
```

図 17　LLVM/Clang-3.9 で検出したエラープログラム

図 18 は表 2 の実験とは別に, GCC-4.5 で検出したエラープログラムを最小化したものである. このプログラムは for 文と if 文を同時に含むため, Orange3 では生成できないものである.

```
01: #include <stdio.h>
02: #define OK()  printf("@OK@\n")
03: #define NG(test,fmt,val)  printf("@NG@ (test = " fmt ")\n",val)
04:
05: int i_max = 0;
06: int j_max = 0;
07: volatile long long e = 1LL;
08:
09: int main (void)
10: {
11:     int i;
12:     int j;
13:
14:     for (i=0; i<i_max; i+=(e/e)) {
15:         for (j=0; j<j_max; j++) {
16:             volatile int u = 1;
17:         }
18:     }
19:
20:     volatile long long x = 2;
21:     volatile int y = 1;
22:     long long t1 = 1LL % x;
23:     int t2 = y % 3;
24:
25:     volatile int c = 0;
26:     volatile int d = 0;
27:     if ( c ) { ; }
29:     else if ( d ) { ; }
30:
31:     if (t1 == 1LL) { OK(); } else { NG("t1", "%d", t1); }
32:     if (t2 == 1) { OK(); } else { NG("t2", "%lu", t2); }
33:
34:     return 0;
35: }
```

図 18　GCC-4.5 で検出したエラープログラム

5. むすび

本稿では, C コンパイラのランダムテストにおいて, 等価変換によりテストプログラムを生成する手法を提案した. 実験の結果, GCC-6.0.0 および LLVM/Clang-3.9 において, 不具合を検出することができた. また, GCC-4.5.0 において, Orange3 では検出できない不具合を検出することができた.

今後の課題としては, 配列, 構造体, 関数呼び出し等の追加により, 不具合検出能力の向上を図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元関西学院大学大学院中橋昌俊氏 (現新明和テクノロジ株式会社), 元関西学院大学由良駿氏 (現防衛省) に感謝致します. また, 本研究を行うにあたり, 御助言や御協力を頂いた, 藤原今日子氏はじめ, 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. 本研究は一部科学研究費補助金 (25330073) による.

(注3)：clang version 3.9.0 (trunk 259289)

(注4)：https://llvm.org/bugs/show_bug.cgi?id=26407

文　献

[1] Plum Hall, Inc.: The Plum Hall Validation Suite for C (online), http://www.plumhall.com/stec.html (accessed 2015-12-03).

[2] Free Software Foundation, Inc.: Installing GCC: Testing (online), http://gcc.gnu.org/install/test.html (accessed 2015-12-03).

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

[4] 日比野佑亮, 石浦菜岐佐: "C コンパイラの算術最適化を対象としたテストスイート CF3," 電子情報通信学会技術研究報告, VLD2014-130 (Jan. 2015).

[5] 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).

[6] X. Yang, Y. Chen, E. Eide, and J. Regehr: "Finding and Understanding Bugs in C Compilers," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 283–294 (June 2011).

[7] W. M. McKeeman: "Differential Testing for Software," Digital Technical Journal, Vol. 10, No. 1, pp. 100–107 (Dec. 1998).

[8] E. Nagai, A. Hashimoto and N. Ishiura: "Reinforcing Random Testing of Arithmetic optimization of C Compilers by Scaling up Size and Number of Expressions," in *Proc. IPSJ Transactions on System LSI Design Methodology*, Vol. 7, pp. 91–100 (Aug. 2014).

[9] K. Nakamura and N. Ishiura: "Introducing Loop Statements in Random Testing of C Compilers Based on Expected Value Calculation," in *Proc. the Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp. 226–227 (Mar. 2015).

[10] 中橋昌俊: "算術式の解析木の導出に基づく C コンパイラのランダムテスト," 関西学院大学大学院理工学研究科情報科学専攻修士論文 (Mar. 2014).

[11] 由良駿, 中橋昌俊, 石浦菜岐佐: "算術式の解析木の導出に基づく C コンパイラのランダムテスト," 電子情報通信学会総合大会, AS-1-4 (Mar. 2015).

[12] V. Le, C. Sun, and Z. Su: "Randomized Stress-Testing of Link-Time optimizers," in *Proc. 2015 International Symposium on Software Testing and Analysis*, pp. 327–337 (July 2015).

[13] V. Le, C. Sun, and Z. Su: "Finding Deep Compiler Bugs via Guided Stochastic Program Mutation," in *Proc. ACM SIGPLAN International Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 386–399 (Oct. 2015).

[14] 日本規格協会: "プログラム言語 C JIS X 3010:2003 (ISO/IEC9899:1999)," (Dec. 2003).