Vol.2013-SE-179 No.11
2013/3/11

モデル検査の反例を用いたC言語プログラムの不具合修正方法の提案と実装

古川直樹[†1]　深海 悟[†1]

プログラムの不具合の検出は主にテストが用いられてきたが，テストだけではすべての不具合を見つけることが難しい．そこで最近モデル検査による検証が用いられるようになり，Magic や CMBC など C 言語を対象としたモデル検査ツールが登場した．モデル検査では，不具合を検知すればその反例を得ることができ，それを確認することで不具合の原因を特定できる．しかし，反例から具体的な修正案を出すツールは存在していない．そこで C 言語プログラムを対象とし，反例から具体的な修正案を出力するツールを開発した．

Auto-suggestion of bug fix method in C programs by model checking counterexamples

NAOKI FURUKAWA[†1]　SATORU FUKAMI[†2]

Conventionally, software testing has been used to find defects within software. But testing can never completely identify all the defects within software. So recently, model checking has come to be used for the discovery of the defect. And model checking tools for C language, for example, Magic and CBMC, has been developed. In the model checking, if a defect is detected we can get a counterexample. And we can determine the cause of the defect by analyzing it. But there is no tool to do this automatically. So we have developed a tool for ANSI C programs to generate a bug fix method automatically based on the analysis of the counterexamples.

1．はじめに

プログラムの不具合の検出は主にテストが用いられてきた．しかし，テストだけではすべての不具合を見つけることは難しい．そこで最近はモデル検査によるプログラムの検証が用いられるようになり，Magic[1]や CMBC[2]など C 言語を対象としたモデル検査ツールが登場してきた．

モデル検査では，不具合を検知すればその反例を得ることができ，それを確認することで不具合の原因を特定できる．また，最近では不具合の原因個所を特定する方法も研究されている[3]．しかし，モデル検査の結果として出力される反例から具体的な修正案を出すツールは存在していない．そこで C 言語プログラムを対象とし，反例から具体的な修正案を出力するツールを開発した．

本論文では，対象とする問題を，C 言語プログラムの配列およびポインタによるメモリ領域の範囲外参照検出およびこれを解消する修正方法の提示にしぼり，モデル検査ツール CBMC から出力される反例をもとに修正方法を提示する方法を提案する．また開発したツールの検証実験の結果について述べる．

検証実験の結果，CBMC の仕様が原因でメモリ範囲外参照の検出ができないケースが存在するが，CBMC が検出した不具合に対しては，多くの場合これを解消する修正方法を正しく提示できることが確認できた．なおこの修正方法は，もともとのプログラムの仕様を考慮したものではないため，仕様に合った修正である保証はない．しかし，不具合箇所の検出ならびにこの不具合を解消する方法の提示は，プログラム開発者にとって大きなヒントになりえることから，本ツールに有効性は高いと考えられる．

2．対象とする問題

プログラムの不具合には，アルゴリズムのミス等により実行結果が予期したものにならないケースと，予期せぬ例外が発生しうるケースがある．本ツールは後者の中の，「配列およびポインタによるメモリの範囲外参照」を対象とする．この不具合は比較的単純だが，これを放置すると意図しない様々な不具合が発生する．また，バッファオーバフローなどの原因につながる．

例えば，図 2.1 のプログラムがこの種の不具合を含んでいる．

†1 大阪工業大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka Institute of Technology.

ⓒ 2013 Information Processing Society of Japan

```
L1： #define NUM 10
L2： int main(void){
L3： int array[NUM];
L4： int *p;
L5： for(p = array; *p != 0; p++){
L6：   scanf(“%d”, p);
L7： }
L8： …
```

図 2.1　メモリの範囲外参照の典型例

図 2.1 のプログラムは配列 **array** の要素に 0 が含まれていない場合，繰り返し処理が無限に行われ，範囲外を参照する．また図 2.2 のプログラムにもこの種の不具合が含まれている．

```
#define NUM 10
…
int array[NUM];
int *p;
for(p = array + NUM; *p != 0; p--){
  scanf(“%d”, p);
}
```

図 2.2　メモリの範囲外参照の典型例その 2

3．解決方法

C 言語を対象とするモデル検査ツールには Magic や CBMC がある．その中で，CBMC はポインタや配列の状態までも検証できるため，本ツールでは CBMC を採用した．CBMC は C 言語プログラムを乗法標準形(CNF)に変換し，SAT 問題を解くことで不具合を網羅的に検証するツールである．

また，本ツールを動作させる開発環境としては，プラグインで拡張ができること，ならびに利用している開発者が多いことから Eclipse を選択した．すなわち，本ツールは Eclipse プラグインで動作するようにした．

本ツールでは，不具合の修正提案は図 3.1 に示す 3 つのフェーズを経て行う．

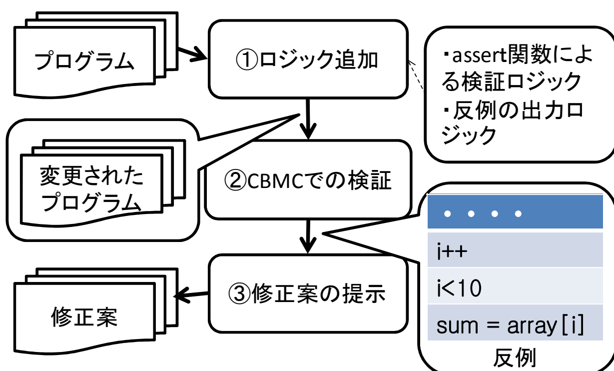

図 3.1　不具合修正までの手順

まず①のフェーズでは，以下の手順に従い，ソースコード上に必要なロジックを追加する．

(1)プログラムの構文解析

BISON･FLEX を用いて構文･字句解析を行い，プログラム全体を図 3.2 の抽象構文木へ変換する．

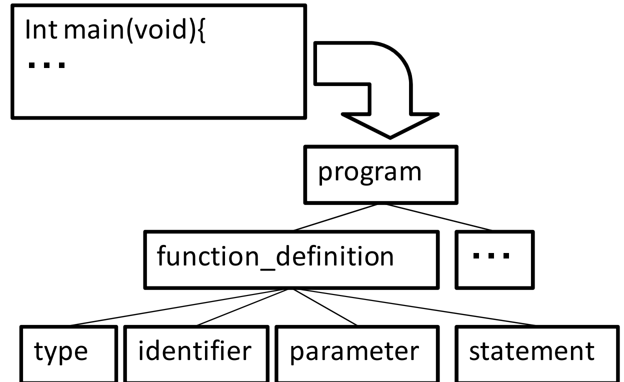

図 3.2　抽象構文木

(2)抽象構文木から変数および型宣言および構造体の情報，**malloc** による動的割り当て，ポインタによる代入式に関した情報を抽出する．

また，ポインタおよび配列の状態を反例で出力・分析するため，各変数に表 3.1 の 4 種類の変数をプログラム中に追加する．なお，これらの変数の値は該当する変数の宣言・動的割り当ておよびポインタによる代入に応じて変更される．

表 3.1 ポインタ・配列の状態を表す変数

max_size	配列の数を示す．もし，変数の場合は **1**，図 2.1 の変数 **array** は **10** になる．
basis_location	ポインタが対象の配列に対して何番目に参照しているか確認するための変数である．図 2.1 の 3 行目で **p=array** で代入した場合は **0**，もしそのあと **p++**した場合は **1** になる．
defined	ポインタにアドレスが代入しているかどうかを示す．**NULL** もしくは未定義の場合は **0**，そうでない場合は **1** になる．
malloc_flag	**malloc** などで動的にメモリ割当てしているかどうかのフラグ．ポインタに動的にメモリを割り当てた場合のみ **1** になる．それ以外は **0** である．

(3)抽象構文木から不具合が発生する箇所（たとえば，配列やポインタ参照を行っている箇所）を探す．そしてその該当箇所に対して表 3.1 の変数を用い，不具合の種類および発生状況を反例で確認するため「**assert** 関数によるロジック」を追加する．

例えば，図 2.1 のプログラムに対しては図 3.3 のロジックを 6 行目(L6)の直前に追加する．

```
if(10 <= 0 + basis_location_1_p){
  エラーメッセージ
  assert(0);
  return 1;
}
```

図 3.3　**assert** 関数による検証ロジック

上記ロジックでは，まず検証対象の変数 **p** は一次元ポインタなので，ポインタで配列の何番目を参照しているかを示す **basis_location_1_p** が **10** 以上なのかを確認する条件式を追加する．

もしこの条件を満たせば，この直後の命令で配列のメモリ範囲外参照が発生することになる．そこでエラーメッセージを出力させたあと，**assert(0)**で強制的にエラーを発生させ，**return 1** で当該関数を強制的に終了するようにする．

この一連の処理を追加することで，その箇所で無限ループを起こして不具合が発生しても，CBMC はそこで CNF の展開が止まり，エラーメッセージを反例として表示することができる．ただし，このアプローチだけではあらゆる無限ループによる不具合を検証することはできない．これについては「4．検証と考察」でふれる．

エラーメッセージに関しては図 3.4 の XML を出力するように追加する．

```
<error type = "Upper Unbound" variable="p">
  <expression value="0"></expression>
  <basis_location value="10">
  </basis_location>
</error>
```

図 3.4　エラーメッセージ

この XML タグはそれぞれ表 3.2，表 3.3，表 3.4 の構成になっている．

また③のフェーズで反例を分析し，不具合の修正案を提示できるようにするため，**printf** 文を用いた「反例の出力ロジック」を追加する．具体的には，反例に含まれる命令およびその命令実行前後の変数の状態を出力できるようにする．たとえば図 2.1 のプログラムの 3 行目の命令(L3)の前に図 3.5 の XML を出力するようにする．また，3 行目の命令（L3）の後に図 3.6 のような XML を出力するようにする．

表 3.2　**error** タグの構造

type 属性	不具合のタイプで，2 種類に分かれる． ①**Upper Unbound**:配列やポインタで配列の上限を超えて参照した場合 ②**Lower Unbound**:配列やポインタで負の番地に参照した場合

表 3.3　**expression** タグの構造

value 属性	不具合を引き起こした配列内の式を示す．たとえば，**p[i]**のようにアクセスした場合は **i** が代入されている．

表 3.4　**basis_location** タグの構造

value 属性	ポインタが対象の配列に対して何番目にアクセスしたかを示す．今回の例では 10 番目にアクセスしたということを示す．

```
<source file="test.c" loc="1" type="expression">
  <prog text="int array [ 10 ];"/>
  <before_variable_info>
    <variable type="int [ 10 ]" name="array"
      value="" max_size="10" basis_location="0"
      defined="1" alloced="0"></variable>
  </before_variable_info>
</source>
```

図 3.5　反例出力ロジックで出力される
実行前の変数の状態例

```
<source file="test.c" loc="1" type="expression">
  <prog text="int array [ 10 ];"/>
    <after_variable_info>
    <variable type="int [ 10 ]"
      name="array" value="" max_size="10"
      basis_location="0"
      defined="1" alloced="0">
    </variable>
  </after_variable_info>
</source>
```

図 3.6　反例出力ロジックで出力される
実行後の変数の状態例

これらの XML はそれぞれ表 3.5，表 3.6 の構成になっている．また，**before_variable** および **after_variable** タグはそれぞれ命令実行前および実行後の変数情報を示し，それぞれは表 3.7 の

ような構成のタグが含まれている．

表 3.5 **source** タグの構造

file 属性	ソースファイル名
loc 属性	命令の位置（行数）
type 属性	命令のタイプ．以下のようなタイプが存在する． **expression**：代入式・単項式 **init_expression**:**for** 文の初期式 **repeat_branch**：**for** 文の継続式 **branch**：**if** 文などの分岐

表 3.6 **prog** タグの構造

text 属性	プログラムの中身
branch_result 属性	分岐の結果条件を満たしている場合は **1**，そうでない場合は **0**

表 3.7 variable タグの構成

type 属性	変数の型
name 属性	変数名
value 属性	変数の値．ただし，配列では何も表示させない
max_size 属性	配列のサイズを示す．
besis_location 属性	ポインタとして配列の何番目に参照しているかを示す
defined 属性	ポインタに変数および配列のアドレスが代入されているかどうかのフラグ
alloced 属性	**malloc** などによって動的にメモリ割り当てを行っているかどうかのフラグ

②のフェーズでは，CBMC で実際に検証し，不具合が確認できれば，不具合のタイプおよび反例が XML として出力される．さらに対象のソースファイルから変数一覧および define で定義されたマクロ一覧を抽出する．

そして③のフェーズで反例から修正案を作成する．不具合の修正案作成はつぎの方法により行っている．

(a) 不具合を回避するための条件を追加する．
(b) 範囲外の原因となる代入を直接修正する．

この 2 通りの方法で修正できる理由について説明する．まずメモリ範囲外参照は用意されている数以上および負の番地にアクセスするパターンしかない．そこで(a)の方法でこれらの条件での実行を回避させることにより，確実にメモリ範囲外参照に陥らないようにすることが可能である．ただし，この方法は図 2.2 のように繰り返しなしで範囲外参照を起こすケースには対応できないため，(b)の方法も併用することで図 2.2 のケースの不具合を回避する．

・(a)の修正方法

(a) の方法による修正案作成は以下の(1)～(3)の手順で行う．

(1)反例の XML から不具合を回避できる条件分岐を探す．すなわち，**type** 属性が**"repeat_branch"** または**"branch"**である **source** タグを探す．

(2)回避できる条件分岐式を見つければ，そこに配列やポインタによるメモリ範囲外参照をできないようにするための条件式を追加する．

たとえば図 2.1 のプログラムに対しては(1)(2)の手順を経て，for 文の継続条件を図 3.7 の下線部のように修正する案を提示する．

```
int array[NUM];
int *p;
for(p = array; p < array + NUM -1 &&
 *p != 0; p++){
scanf(“%d”, p);
}
```

図 3.7 メモリの範囲外参照の修正例

ただし，回避する条件分岐が **else 文**ブロックを含まない **if** 文ブロックであり，なおかつその命令がループ内で実行されている場合は，図 3.8 のように **if** 文ブロックのあとに **else 文**ブロックを追加する．

```
else{
  break;
}
```

図 3.8 追加する else 文

(3)回避する条件分岐式が見つからなければ，不具合が発生した命令に対して **if** 文ブロックに挟まれるように条件式を追加する．すなわち，図 3.9 のように修正する．

```
                  if(i < NUM){
a[i] = x;   ⇒       a[i] = x;
                  }
```

図 3.9 手順(3)での修正例

ただし不具合の発生した命令がループ内で実行されている場合は図 3.8 の **else** 文ブロック命令を **if** 文ブロックの後ろに追加する.

・(b)の修正方法

(1)エラーで発生した配列の最大数を **variable** タグの max_size 属性から取得する.
(2)マクロ一覧から配列の最大数に近い値を持ったマクロを調べる.
(3)見つかれば，そのマクロを用いて修正し，見つからなければ新しいマクロを定義し，新しいマクロ**-1** を代入する．新しいマクロ名についてはユーザが決める．また，配列の式が i+1 といった場合は，代入式に対してさらに -1 を追加する.

たとえば，図 2.2 のプログラムに対して，図 3.10 の下線部のように修正を行う.

```
#define NUM 10
…
int array[NUM];
int *p;
for(p = array + NUM - 1; *p != 0; p--){
  scanf(“%d”, p);
}
```

図 3.10 メモリの範囲外参照の修正例その 2

以上で述べた(a)，(b)2 通りの方法で修正方法提案を生成するが，状況によって修正方法を変更する必要がある．まず反例がつぎつぎの条件を満たしているか否かで修正方法を変更しなければいけない.

・最後に実行された条件分岐の後に配列にかかわる変数の値が変更されているかどうか

この条件が成り立っているかを判断するため，まず反例の XML からポインタおよび配列参照で使用された変数一覧を取り出す．たとえば，**a[i+j]** の場合は変数 **i**，**j** について取り出す．そしてそれらの変数が代入式などで変更されているかどうかチェックする．すなわち，該当変数の情報を含んだ **after_variable** タグに含む **source** タグを探す.

もし，見つかれば見つかった代入式に(b)の修正方法を用い，見つからない場合は最後に実行された条件分岐に(a)の修正方法を用いる．もしこの条件を満たした状態で，最後に実行された条件分岐に(a)の修正を行ったら，そのあとの代入式で範囲外参照を引き起こす代入が行われる可能性があり，(a)による修正の意味がなくなるからである.

また，反例がつぎの条件を満たせば，(b)の修正方法も用いる.

・**for** 文の初期式の後からエラー発生までにおいて，**for** 文の初期式の代入先の変数が変更されなかった場合

これを満たすことは繰り返しなしでメモリ範囲外参照を起こすことになる．たとえば，図 2.2 の例がこれを満たす．この場合は状況によって(b)の修正が好ましい場合がある．とくに図 2.2 の場合は(b)の修正が好ましい.

なお，いずれの修正方法においても，エラーが発生した配列がポインタ経由で参照されていて，そのポインタの参照元がたどれない場合（たとえばそのポインタが関数の引数だった場合）は，別途ポインタ変数を関数内に新たに宣言し，そのポインタ変数を用いて修正する.

たとえば，図 3.11 の配列の合計値を求める **sumArray** 関数に対して，図 3.12 または図 3.13 のように修正する．なお，図中に存在する **orig_ptr** はユーザ側で名前を決めた新しいポインタ変数である.

```
#include<stdio.h>
#define NUM 10

int sumArray(int *a){
  int sum = 0;
  while(*a != 0){
    sum += *a;
    a++;
  }
  return sum;
}
int main(void){
  int i;
  int sum;
  int a[NUM];
  for(i = 0; i < NUM; i++)
    scanf("%d", &a[i]);
  sum = sumArray(a);
  printf("合計値:%d",sum);
}
```

図 3.11 メモリ範囲外参照を含んだプログラム

```
int sumArray(int *a){
  int *orig_ptr = a;
  int sum = 0;
  while(*a != 0){
    if(a < orig_ptr + NUM ){
      sum += * a ;
    }
    else{
      break;
    }
    a++;
  }
  return sum;
}
```

図 3.12 **sumArray** 関数の修復例その 1

```
int sumArray(int *a){
  int *org_ptr = a;
  int sum = 0;
  while(a < org_ptr + NUM && * a != 0){
    sum += *a;
    a++;
  }
  return sum;
}
```

図 3.13 **sumArray** 関数の修復例その 2

4．検証と考察

本章では，開発したツールを使って実際にプログラムを検証・修正を行い，正しい修正が行われたかどうか，行われなかったらなぜそうなったのかを検討する．

4．1 検証・修正に成功したケース

本ツールを使用し，メモリの範囲外参照を含みやすい事例を取り上げ検証した．まず検証・修正に成功したケースについて述べる．今回は繰り返しおよび配列の参照・代入処理を多用する図 4.1 に示すバブルソートプログラムに適用した．

図 4.1 のプログラムでは 7 行目の **j=NUM;**の代入によって，繰り返しなしで範囲外を参照する不具合が発生する．また，この関数を呼び出す **main** 関数は図 4.2 に示す．

これを検証した結果，図 4.1 の 8 行目の **a[j]** 部分において配列 **a** の **10** 番目に参照しようとした旨のメッセージが発生し，7 行目の **j=NUM;**を **j=NUM-1;**に修正する方法と，8～11 行目を図 4.3 のように修正する方法の 2 通りが提示された．これら修正案に従って修正し再度検証した結果，いずれの修正案にしたがった場合も不具合は発生しなかった．そして，実際に実行してみたところ，前者は正常に実行できたものの，後者の修正では正常に実行できなかった．その理由は，後者の修正では肝心なソート処理が一切行われないことになるためである．

```
#include<stdio.h>
#define NUM 10
int sortArray(int a[NUM]){
  int *ptr = a;
  int i,j;
  for(i=0;i<NUM-1;i++){
    for(j = NUM ;j>i;j--){
      if(a [ j ] < a [ j - 1 ]){
          int t=a[j];
          a[j]=a[j-1];
          a[j-1]=t;
      }
    }
  }
  return 0;
}
```

図 4.1 不具合を含むソートプログラム

```
int main(void){
  int i;
  int a[NUM];
  for(i = 0; i < NUM; i++)
    scanf("%d", &a[i]);
  printf("ソート前:");
  for(i = 0; i < NUM; i++) printf("%d ",a[i]);
  i = sortArray(a);
  printf("\n ソート後:");
  for(i = 0; i < NUM; i++) printf("%d ",a[i]);
  return 0;
}
```

図 4.2 ソートプログラムを呼び出す **main** 関数

```
if(j < NUM && a [ j ] < a [ j - 1 ]){
  int t=a[j];
  a[j]=a[j-1];
  a[j-1]=t;
}else{
  break;
}
```

図 4.3 ソートプログラムに対する修正案

つぎに，図 4.1 のプログラムを図 4.4 のように変更して再度検証した．このプログラムは 7 行目の最初の代入式は正常になる代わりに，継続条件を敢えて **a[j]!=0** にすることで，**0** を含まない配列という条件で無限ループが発生し，メモリ範囲外参照不具合が発生する．

```
#include<stdio.h>
#define NUM 10
int sortArray(int a[NUM]){
  int *ptr = a;
  int i,j;
  for(i=0; i<NUM-1; i++){
    for(j = NUM-1 ;a[j]!=0;j--){
      if(a [ j ] < a [ j - 1 ]){
          int t=a[j];
          a[j]=a[j-1];
          a[j-1]=t;
      }
    }
  }
  return 0;
}
```

図 4.4　別の不具合を含むソートプログラム

このプログラムを検証した場合，図4.4の8行目で **a[j-1]**で配列 **a** の**-1** 番目に参照しようとした旨のメッセージが発生し，このプログラムの 8～11行目に対して図4.5のような修正案を表示することができた．その修正案に従って修正し，再度検証した結果不具合は発生しなかった．また，このプログラムを実際に実行した結果，正常に実行することができた．

これらのことから，本ツールで提示した修正案はいずれも発生した不具合は修正できたものの，すべての修正案がプログラム的に正しい修正とはいえないことがわかる．

```
if(j - 1 >= 0 && a [ j ] < a [ j - 1 ]){
  int t=a[j];
  a[j]=a[j-1];
  a[j-1]=t;
}else{
  break;
}
```

図 4.5　修正案

４．２　検証に失敗したケース

ここでは検証に失敗したケースについて述べる．これらはいずれも CBMC 側で検証に至らなかったケースである．

(1)無限ループ内で繰り返しに依存しない不具合が存在する場合

たとえば，図4.6のプログラムがこれに該当する．

```
#include<stdio.h>
#define NUM 10
int sortArray(int a[NUM]){
  int *ptr = a;
  int i,j;
  for(i=0;  a[i]!=0;  i++){
    for(j = NUM-1 ;j>i;j--){
      if(a [ j ] < a [ j - 1 ]){
          int t=a[j];
          a[j]=a[j-1];
          a[j-1]=t;
      }
    }
  }
  return 0;
}
```

図 4.6　検証に失敗したソートプログラム

このプログラムの 6 行目は変数 **i** に依存して繰り返し処理を行うが，不具合を起こす可能性のある 8 行目の部分は変数 **j** しか依存していない．そのため，CBMC の CNF 生成時に，8 行目で不具合を起こす状態に辿りついても，6 行目の繰り返し処理部分で式を無限に展開し続けるため，検証に至らなかった．

(2)検証対象の配列がプログラム引数だった場合

たとえば，図 4.2 のプログラムを変更した図 4.7 が１つの例である．

```
int main(int argc, int *argv[]){
  int i;
  int *array = malloc(sizeof(int)*argc);
  for(i = 0; i < argc; i++)
    a[i] = atoi(argv[i]);
  printf("ソート前:");
  for(i = 0; i < argc; i++)
    printf("%d ",array[i]);
  i = sortArray(array);
  printf("\n ソート後:");
  for(i = 0; i < argc; i++)
    printf("%d ",array[i]);
  return 0;
}
```

図 4.7　プログラム引数を用いたソートプログラムの **main** 関数部分

プログラム引数は実行コマンドに応じて数が変化する．そのため，実行コマンドを固定しないかぎり，表 4.1 の配列の状態を表す変数が使えず，正常に検証できない．逆に言えば，実行コマンドを固定にすれば検証は可能になるが，あらゆる実行コマンドで実行しないと本来のモデ

ル検査の網羅性が損なわれてしまう．たとえば，プログラム引数が 1 個なら問題ないが，3 個の場合どうなるのか，あるいはその引数の値自体は数字でなく文字列といった不正なパターンが入っても問題ないのか検証しなければならない．

そのため，ユーザに実行コマンドを複数指定することでモデル検査の網羅性を犠牲にしても不具合を検証しなければいけないのか，あるいは実行コマンドを何らかの手段で自動的に生成して検証させる方法を考え確立させるべきなのか考えなければならない．この点に関しては今後の課題になる．

５．結論

モデル検査から出力される反例に着目し，不具合の修正案を提示するツールの開発を行った．具体的には，C 言語プログラムにおいてバッファオーバフローなどのさまざまな不具合の原因にもつながる配列・ポインタによるメモリ範囲外参照を検出し，この不具合を解消する修正提案を出力する方法の提案ならびにこの方法を実装したツールの開発を行った．

さらに，繰り返しや配列参照を多用するバブルソートのプログラムにさまざまなメモリ範囲外参照を引き起こす不具合を埋め込み，このツールによる検証を行った．その結果，CBMC の制約が原因で，メモリ範囲外参照のあらゆるパターンには対応できなかったものの，典型的なケースでは正常に不具合を指摘し，修正提案を提示することが確認できた．CBMC による制約によって検証できなかった部分に関しては別のアプローチで実装するなり対策を取らなければならない．

本ツールが出力する修正提案は，当然のことながら開発者の意図（プログラムの仕様）を反映したものではない．しかし，不具合を回避する方法を提示することは，正しい修正のための大きなヒントになりえると考えている．

なお，このツールの使い方としては，関数単位での検証を想定している．というのも，関数単位で検証した場合は検証範囲も狭まるため，検証時間も短く，出力された修正提案の妥当性も容易に確認できると想定されるからである．

【参考文献】

[1] Chaki,S., Clarke,E.M., et.al., Modular Verification of Software Components in C, IEEE Tr. on SE, Vol.30, No.66, pp.388,2004.
[2] Clarke,E.M., et.al, A Tool for Checking ANSI-C Programs, LNCS Vol.2988, pp.168, 2004.
[3] 青木，松浦，モデル検査を用いたプログラムにおける再現性の低い潜在的欠陥の抽出手法，信学技報， vol.110， no.468，KBSE2010 47-60，pp 79-84，2011.