

モデル検査による C 言語プログラムの不具合修正ツールの開発

古川直樹 深海 悟

大阪工業大学大学院 情報科学研究科

1. はじめに

プログラムの不具合の検出には、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が難しい。そこで 2000 年ごろからモデル検査による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箇所を特定する方法も研究されている[3]。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。

2. 対象の問題

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想したものにならないケースと、予想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、バッファオーバーフローなどの原因につながる。

例えば、図 1 のプログラムがこの種の不具合を含んでいる。

```
int array[10];
int *p;
for(p = array; *p != 0; p++){
    scanf("%d", p);
}
```

図 1 メモリの範囲外参照の典型例

図 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。

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 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、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 2 に示す 3 つのフェーズを経て行う。

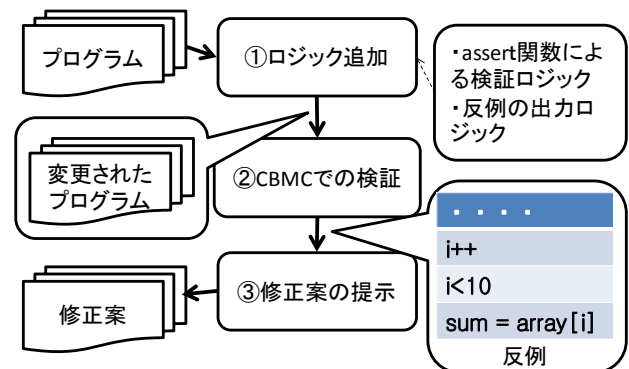


図 2 不具合修正までの手順

まず①のフェーズでは、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対象プログラムに追加する。例えば、図 1 のプログラムに対しては図 3 のロジックを 4 行目の直前に追加する。

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

図 3 assert 関数による検証ロジック

また③のフェーズで反例を分析するために、printf 文を用いた「反例の出力ロジック」を追加する。このロジックで実行する命令および実

Development of the bug fix tool by Model checking in C

Naoki FURUKAWA, Satoru FUKAMI

Graduate School of Information Science and Technology, Osaka Institute of Technology.

行前後の状態を出力できるようにする。たとえば図1のプログラムの1行目の前後で図4のXMLを出力するようにする。

```
<source file="test.c" loc="1"
type="expression">
<prog text="int array [ 10 ]:"/>
実行前または後の状態
</source>
```

図4 実行する命令の情報

図4の「実行前または後の状態」の部分にはそれぞれ図5のXMLが出力される。

```
実行前の状態
<before_variable_info>
<variable type="int [ 10 ]" name="array"
max_size="10" ...></variable>
</before_variable_info>

実行後の状態
<after_variable_info>
<variable type="int [ 10 ]" name="array"
max_size="10" ...></variable>
</after_variable_info>
```

図5 実行前および実行後の変数の状態

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。

そして③のフェーズで反例から修正案を提案する。ここで、不具合の修正はつぎの方法により行っている。

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

前者は反例を逆順に辿り、不具合を回避できる条件分岐を見つけ、範囲外を参照しないような条件式を追加する。たとえば図1のプログラムに対しては3行目を下線部のように修正する。

```
for (p = array; p < array + 10 && *p != 0; ...
```

後者は、添え字やポインタなど配列にかかわる変数に対する代入式を見つけて修正する。たとえば、図6の式の代入式 `i = NUM;` に対して、`i = NUM - 1;`へ修正する。

ただし、修正に新たな変数を作成しなければ

いけないときは、変数名をユーザで決める。

```
#define NUM 10
...
for(i = NUM; i >= 0; i--)[
    array[i] = i;
```

図6 繰り返しなしに範囲外の参照を行う例

4. 検証と考察

本ツールをメモリの範囲外参照を含んだソートプログラムに適用した。繰り返しなしで範囲外を参照するケースや無限ループで範囲外参照を起こすケースは正しく修正提案が出力された。しかし、以下のケースはCBMCのCNF生成時に、繰り返し処理部分で式を無限に展開し続けるため、検証できなかった。

- ・無限ループ内で繰り返しに依存しない不具合が存在した場合
- ・検証対象の配列がプログラム引数だった場合

5. 結論

本ツールでは、CBMCの制約から、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が、典型的なケースでは正常に不具合を指摘し、修正提案を出力することができた。本ツールで修正できなかった不具合に関しては今後の課題となる。

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 Transactions on SE, Vol.30, No.6, 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.