SRA の技術情報誌
GS LetterNeo
Vol.154　2021 年 5 月

C/C++プログラム向け検証系 CBMC

熊澤 努

DX 技術本部 先端技術研究所

はじめに

CBMC[1]は C/C++言語で書かれたプログラムを対象とする自動検証ツールです。Windows、Linux、MacOS 向けのバイナリが公開されていて、無償で使うことができます。CBMC の特徴は、NULL ポインタへのアクセスや配列の境界超過などの発見の難しい不具合を、プログラムを実行することなく検出する点にあります。本稿では、CBMC の主要な機能を実際に動かしてみます。

CBMC をインストールする

Windows 10 場合、CBMC をインストールする方法は 2 通りあります。1 つは、Visual Studio 向けプラグインを導入する方法、もう 1 つは Linux 仮想環境 WSL(Windows Subsystem for Linux)にインストールする方法です。今回は後者を採ることにし、Ubuntu 18.04 LTS に CBMC をインストールします（WSL の環境を構築する手順は省略します）。WSL のターミナルを起動して、以下のコマンドを実行してください。CBMC が自動的にインストールされます。今回インストールしたバージョンは CBMC 5.6 です。

```
> apt-get install cbmc
```

[1] https://www.cprover.org/cbmc/

ポインタの検査をする

CBMC を使った検証を簡単な例で試してみましょう。下のプログラムを file1.c という名前で保存します。

file1.c

```
#include<stdio.h>

int main(void) {
    int *p;
    int x;
    p = NULL;
    x = *p;
    return 0;
}
```

まず、上のプログラムを実行して結果がどうなるか見てみます。gcc（必要に応じてインストールしてください）を使って file1.c をコンパイルした後に実行すると、次のように、実行時エラーが発生します。

```
> gcc file1.c -o file1.out
> ./file1.out
Segmentation fault (core dumped)
```

このエラーは、7 行目で変数 x が NULL ポインタ p の指す先を参照したために発生しました。C/C++言語のプログラムのポインタに関係するエラーは、発見が難しいことで知られています。次に、このエラーを CBMC で検出してみます。CBMC は cbmc コマンドで実行します。ポインタの検査を行うには、検査対象である file1.c とともに、オプション --pointer-check を指定します。

```
> cbmc file1.c --pointer-check
CBMC version 5.6 64-bit x86_64 linux
Parsing file1.c
…(略)…

** Results:
[main.pointer_dereference.1] dereference failure: pointer NULL in *p: FAILURE
[main.pointer_dereference.2] dereference failure: pointer invalid in *p:
SUCCESS
[main.pointer_dereference.3] dereference failure: deallocated dynamic object
in *p: SUCCESS
[main.pointer_dereference.4] dereference failure: dead object in *p: SUCCESS

** 1 of 4 failed (2 iterations)
VERIFICATION FAILED
```

Results 以降が検査結果です。ポインタ検査では Results のすぐ下に表示された 4 項目を検査していて、そのうち 1 番上の検査項目についてエラーを検出しました。検出したエラーはポインタ p が NULL のため、その指す先の値を参照する(dereference)ことができない、

というものです。2 番目から 4 番目はそれぞれ、無効な領域をポインタが指していないか、解放済みの領域を指していないか、解放されずに残存する領域をさしていないか、という項目を検査しています。最終行の VERIFICATION FAILED は、ポインタの検査の結果、エラーを検出したことを表しています。

--trace オプションをつけて検査を実行すると、より詳細な検査情報も出力することができます。この詳細情報に従ってプログラムを順に 1 行ずつ辿っていくと、7 行目でエラーになることが分かります。

```
> cbmc file1.c --pointer-check -trace
…(略)…

Trace for main.pointer_dereference.1:

State 18 file file1.c line 4 function main thread 0
------------------------------------------------------
  p=((signed int *)NULL) (0000000000…略…000000000)

State 19 file file1.c line 5 function main thread 0
------------------------------------------------------
  x=0 (00000000000000000000000000000000)

State 20 file file1.c line 6 function main thread 0
------------------------------------------------------
  p=((signed int *)NULL) (0000000000…略…000000000)

Violated property:
  file file1.c line 7 function main
  dereference failure: pointer NULL in *p
  !(POINTER_OBJECT(p) == POINTER_OBJECT(((signed int *)NULL)))

…(略)…
```

配列の境界検査をする

CBMC は配列の境界を越えてアクセスしたかどうかを検査することもできます。次のプログラムを考えます(file2.c とします)。

file2.c

```c
int main(void) {
    int i;
    int buf[10];
    for(i = -1; i <= 10; i++) {
        buf[i] = 0;
    }
    return 0;
}
```

3 行目で長さ 10 の配列を宣言し、4 行目から 6 行目でその各要素を初期化しています。C 言語では配列の添え字は 0 から 9 としなければなりませんが、このプログラムは、-1 から 10 までと配列の境界を越えてアクセスしています。そのため、コンパイルには成功しますが、実行時エラーが発生します。

```
> gcc file2.c -o file2.out
> ./file2.out
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

CBMC で境界検査を行うには、--bounds-check オプションを指定します。なお、さらに --trace オプションをつけると、for 文を展開した詳細な検査情報を出力することができます。

```
> cbmc file2.c --bounds-check
…(略)…
** Results:
[main.array_bounds.1] array `buf' lower bound in buf[(signed long int)i]:
FAILURE
[main.array_bounds.2] array `buf' upper bound in buf[(signed long int)i]:
FAILURE

** 2 of 2 failed (1 iteration)
VERIFICATION FAILED
```

CBMC は配列にアクセスの下限と上限の両方について、配列の有効範囲内かどうか検査します。今回は下限が-1、上限が 10 のため、どちらもエラーが検出されています。

アサーションの検査をする

C/C++言語には、アサーション(不変表明)という実行時にエラーを検出する仕組みが備えられています。アサーションには、プログラム実行時に成立してほしいことを条件式の形式で自由に書くことができます。プログラマがアサーションを書いておくと、その箇所をプログラムが実行中実行する際に、条件式を計算します。計算結果が真の場合は実行を継続しますが、偽であるとプログラムを停止します。

CBMC は、プログラムを実行せずにアサーション違反を検査します。次のプログラム file3.c を考えます。

file3.c

```c
#include<stdio.h>
#include<assert.h>

int main(void) {
    int *p;
    assert(p != NULL);
    return 0;
}
```

5 行目でポインタ p を宣言し、6 行目でアサーションを使って p が NULL であることを要請しています。しかしながら、p を特定の値に初期化していないため、その値はプログラムを実行してみないと分かりません。実際、プログラムを実行すると、アサーション違反になって 6 行目で停止します。

```
> gcc file3.c -o file3.out
> ./file2.out
file3.out: file3.c:6: main: Assertion `p != NULL' failed.
Aborted (core dumped)
```

アサーションを CBMC で検査する際にはオプションは不要です。検査結果から、CBMC が違反を検出していることが分かります。このエラーは、p が NULL の場合にはアサーションに違反していることを意味しています。

```
> cbmc file3.c
…(略)…
** Results:
[main.assertion.1] assertion p != NULL: FAILURE

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

今度は、file3.c のアサーションを以下のように書き換えて検査してみましょう(file4.c とします)。file3.c とは条件式の真偽を逆にしてるので、今度は検査しても違反にならないようにも思えますがどうでしょうか。

file4.c

```
#include<stdio.h>
#include<assert.h>

int main(void) {
    int *p;
    assert(p == NULL);
    return 0;
}
```

CBMC で file4.c を検査した結果を以下に示します。

```
> cbmc file4.c
…(略)…
** Results:
[main.assertion.1] assertion p == NULL: FAILURE

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

やはりエラーとなりました。今回は、p が NULL でない場合にアサーションが偽であるというわけです。CBMC はポインタ p の値が NULL の場合と NULL ではない場合のどちらの場合についてもエラーを検出することができます。このように、考えられる全ての状況を検

査する方式を網羅検査といい、初期化していない変数の検出のような、プログラムの実行に依存する潜在的なエラーの検出において力を発揮します。

関数を検査する

次に、自作の関数を検査してみましょう。例として、指定された数の逆数を求める関数 inverse を検査することを考えます(fil5.cpp)。

file5.cpp

```
#include<assert.h>

void inverse(float x, float &y) {
    y = 1.0f / x;
    assert(y >= 0);
}
```

変数 x の値が 0 のときには 0 での割り算となり、関数 inverse は計算ができません。また、アサーションを見ればわかるように、計算結果を格納して呼び出し元に渡す変数 y の値は 0 以上であるとプログラマは考えています。しかし、x の値は負かもしれないので、その点は保証はされていません。

CBMC での関数 inverse の検査は、オプション--function をつけて関数名 inverse を指定することで実行できます。また、ゼロ除算が発生するかどうかを検査するために、オプション--div-by-zero-check もつけることにしましょう。以下のように、ゼロでの除算とアサーション違反の両方を検出することができました。

```
> cbmc file5.cpp --function inverse --div-by-zero-check
…(略)…
** Results:
[inverse.division-by-zero.1] division by zero in 1.000000f / x: FAILURE
[inverse.assertion.1] assertion y >= 0: FAILURE

** 2 of 2 failed (2 iterations)
VERIFICATION FAILED
```

さて、別の検証で変数 x の値が 0 より大きい値を取ることが分かっているときはどうでしょうか。その場合、ゼロ除算もアサーション違反も発生しないはずなので、上では無駄な検査をしていることになります。そこで、次のように x が 0 より大きいという検査の前提条件を追加します。

file6.cpp

```
#include<assert.h>

void inverse(float x, float &y) {
    __CPROVER_assume(x > 0);
    y = 1.0f / x;
    assert(y >= 0);
}
```

__CPROVER_assume をつけると、CBMC は条件式 x > 0 に反する結果を無視します。このファイル fil6.cpp を検査してみます。

```
> cbmc file6.cpp --function inverse --div-by-zero-check
…(略)…
** Results:
[inverse.division-by-zero.1] division by zero in 1.000000f / x: SUCCESS
[inverse.assertion.1] assertion y >= 0: SUCCESS

** 0 of 2 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

成功、つまり、違反がないという結果になりました。このように、CBMC には、ユーザにとって不要な検査をしないようにする機構も組込まれています。

おわりに

今回は、C/C++言語で書かれたプログラムを静的に検証するツール CBMC を使って、簡単なプログラムの検証を行いました。CBMC には、今回紹介した以外にも、数値のオーバーフロ-の検査など多様な検査機能が実装されています。また、Java のバイトコードを対象とした検査ツール JBMC[2]も開発されています。CBMC は網羅検査をするため、実行時エラーとなってはじめて気づくような難しい問題を発見する有力なツールの一つです。小規模な関数の検証から始めるなど、気軽に使ってもらえればと思います[3]。

[2] http://www.cprover.org/jbmc/

[3] 以下の文献には、Amazon Web Services (AWS)における開発での CBMC の適用事例が詳しく記述されています。

Nathan Chong *et al.* : Code-level model checking in the software development workflow at Amazon Web Services, *Software: Practice and Experience*, 2021; 51: 772 – 797. https://doi.org/10.1002/spe.2949.

GSLetterNeo Vol.154
2021 年 5 月 20 日発行
発行者　株式会社 SRA 先端技術研究所

編集者　土屋正人　熊澤努　方 学芬
バックナンバー http://www.sra.co.jp/gsletter
お問い合わせ　gsneo@sra.co.jp

株式会社SRA
〒171-8513 東京都豊島区南池袋 2-32-8

夢を。Yawaraka Innovation
やわらかいのべーしょん