CBMCを用いたC言語関数の形式仕様の導出

岡部 竜
Tatsuya OKABE

北九州市立大学大学院　国際環境工学研究科
情報工学専攻　山崎進研究室
The University of KitaKyushu

背景

- ソースコード品質(武市氏による定義[1])
  - 動作に誤りがない
  - 効率的に動作する
  - 保守しやすい
- ソースコード品質の保証

➡コードについて正確な理解が必要

- コード理解のためには仕様書

➡理解のために十分な仕様書がある事の方が少ない

コードから直接仕様を読み取れる技術が必要

[1] 武市正人，”構文解析を用いたCOBOLソースコード品質点検ツールの開発”，exa review No.10, 2010

目的

- C言語のモデル検査ツールCBMCを用いて
  コード中の一関数の論理構造を求め
  その論理構造から関数の事後条件を導く

CBMCとは

- C/C++コードを対象とするモデル検査ツール
- 有界モデル検査法による検証
  - ポインタ検証や配列境界の検査
  - ユーザー定義のアサーションについての検証
- 検証するコードは完成されている必要はなくコードに含まれる1つの関数についてのみの検証が可能
- 検証のためCBMC内部でコードを**静的単一代入(SSA)形式表現に変換**
  - **CBMCにはこれを出力する機能**

➡SSA形式表現を事後条件の導出に利用

事後条件の導出

- 簡単な処理を行う関数を含むC言語コードを用意
- CBMCを用いてC言語関数のSSA形式を求め，そこから関数の事後条件を導く
  - `cbmc sample.c --function divide --unwind 3 --show-vcc`

```
void divide(int x, int y)
{
    int r, q;
    r = x;
    q = 0;

    while(r > y){
        r = r - y;
        q = q + 1;
    }
}
```

図1，サンプルコード

事後条件の導出

```
file sample01.c line 8 function function
unwinding assertion loop 0
{-1} __CPROVER_deallocated#1 == NULL
{-2} __CPROVER_malloc_object#1 == NULL
{-3} __CPROVER_malloc_size#1 == 0
{-4} __CPROVER_malloc_is_new_array#1 == FALSE
{-5} __CPROVER_rounding_mode#1 == 0
{-6} x!0@1#1 == nondet_symbol(symex::nondet0)
{-7} y!0@1#1 == nondet_symbol(symex::nondet1)
{-8} r!0@2#1 == x!0@1#1
{-9} q!0@2#1 == 0
{-10} \guard#1 == !(y!0@1#1 >= r!0@2#1)
{-11} r!0@2#2 == r!0@2#1 + -y!0@1#1
{-12} q!0@2#2 == 1
{-13} \guard#2 == !(y!0@1#1 >= r!0@2#2)
{-14} r!0@2#3 == r!0@2#2 + -y!0@1#1
{-15} q!0@2#3 == 2
{-16} \guard#3 == !(y!0@1#1 >= r!0@2#3)
{-17} r!0@2#4 == r!0@2#3 + -y!0@1#1
{-18} q!0@2#4 == 3
|------------------------
{1} !(\guard#1 && \guard#2 && \guard#3)
```

事後条件の導出に必要な部分

図２，コードのSSA形式表現

事後条件の導出

- 添字が大きい変数に対して，添字の小さいものを順次代入していく

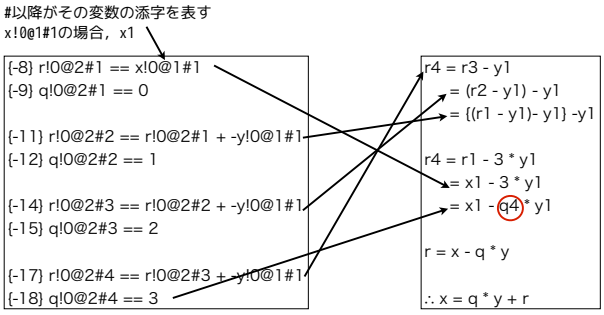

図３，事後条件導出の様子

事後条件の導出

- 得られた関数の事後条件は **x = q * y + r**
  - 入力 x，y に対して関数が実行された時 x = q * y + r が成り立たなければならない
  - 被除数 x，除数 y，商 q，余 r で表される除法を実現する関数である事が分かる

実験

- オープンソースのC言語コードを用い，実際にコード中の一関数について事後条件を求める
- 対象とした関数：**pwm_start()**
  - リアルタイムOS Nuttx のデバイスドライバ pwm.c 中の一関数
    - PWM生成のリクエストがなされた時，PWM生成の条件が満たされている事を確かめる
    - 条件が満たされていない時は条件が満たされるまで待機する
  - pwm.c
    - NuttX によるPWMの操作のための種々の関数が記述されたコード

実験

- **pwm_start()**
  - LOC(『;』『{』による)：**20**
  - 生成されたSSA形式表現：**51**

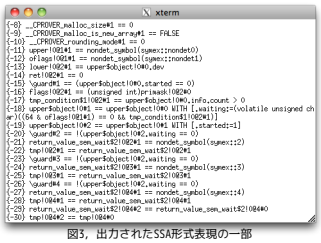

図3，出力されたSSA形式表現の一部

導出した事後条件(論理式)

{(O_NONBLOCK ∧ oflags) ==0} ∧ {(upper -> info.count) > 0} ∧ (upper -> sterted == 1)

結果と考察

- 手がかりがコードしかないという状況でコード中の一関数の事後条件を求めることができた
  - 論理的表現による事後条件
  - SSA形式表現が出力されてからは，添字にそって代入を繰り返すという操作だけ

検討課題

- 規模の大きいプログラムについて事後条件を求めるとき，膨大な量のSSA形式表現が出力され，それらを扱わなければいけない
  - **作業量の大幅な増加**
- 条件分岐などによって変数の状態などを一意に定められない場合が発生する
  - **コード中またはSSA形式表現中の事後条件の導出に直接影響を与えない部分の削減**

課題解決のアイデア

- **SSA形式上での表現の最適化**
- **コードまたはSSA表現形式のスライシング**

これらの技術について調査，適用していく事で
CBMCによる事後条件導出の簡単化を目指す

今後の展望

- 本手法にはどのような利用法が考えられるか
- **案1：ソースコード理解の手がかり**
  - 正常終了したときの条件は分かる
  - 形式仕様の旨みが活かせない
- **案2：VDMなど人間が与えた事後条件との比較，評価**
  - 導出したものが，一般的な事後条件としての要件を満たしているか etc.
- **その他の有用な利用法について調査，検討する**