Falsification-driven Verification

長谷川 央

はじめに

ソフトウェアモデル検査という技術があり、以下のようなツールが知られている

- CBMC (Bounded Model Checker for C and C++ programs)
- The Kani Rust Verifier

これらのモデル検査器は、入力としてソースコードを受け取り、
モデル検査を行い、
範囲外アクセスなどのよくあるバグや、
ユーザ側が指定した性質の違反がないかを
確認してくれる

```
#include "sort.h"
int a[SIZE];
int ref[SIZE];
int nondet_int();
int main () {
  int i, v, prev;
  int s = nondet_int();
  __CPROVER_assume((s > 0) && (s <=SIZE));
  for (i = 0; i < s; i++) {
    v = nondet_int();
    printf ("LOG: ref[%d] = %d\n", i, v);
    ref[i] = v; a[i] = v;
  }
  sort(a, s);
  prev = a[0];
  for (i = 0; i < s; i++) {
    printf ("LOG: a[%d] = %d\n", i, a[i]);
    assert (a[i] >= prev);
    prev = a[i];
  }
}
```

Groce, A., Ahmed, I., Jensen, C. *et al.* How verified (or tested) is my code? Falsification-driven verification and testing. *Autom Softw Eng* **25**, 917–960 (2018). https://doi.org/10.1007/s10515-018-0240-y

問題 (1/2)

…このコードで何が検証できたのか？

以下のassert文で、ソートが正しく行われたことが保証できているのか？

```
for (i = 0; i < s; i++) {
  printf ("LOG: a[%d] = %d\n", i, a[i]);
  assert (a[i] >= prev);
  prev = a[i];
}
```

問題 (2/2)

実は以下のケースに対応できていない

入力 : [10,3,25,82]

期待されるソート結果 : [3,10,25,82]

誤って通ってしまう出力1 : [0,0,0,0]

誤って通ってしまう出力2 : [1,2,3,4]

検証のやり方

前述のような問題を防ぐために、以下のようなアプローチで検証は行われる

1. 検証用のコードを書く
2. バグを意図的に仕込む
3. 検証用のコードでバグが検知できることを確かめる

しかし、この方法では、検証の正しさが結局、発想力に依存してしまっている

2つ目のステップを自動化したい...

→ Falsification-driven Verification

Falsification-driven Verification

mutationの技術を使って、ランダムにコードを変更しバグを仕込む

1. 検証用のコードを書く
2. mutationを行い、検証対象のコードをランダムに変更する
3. mutation後のコードに対して検証を実行する
4. 検証が成功してしまったコードを目視で確認する
5. バグがあるのに検証に成功しているようなら、
   そのバグを検知できるように検証用のコードを修正する

Groce, A., Ahmed, I., Jensen, C. *et al.* How verified (or tested) is my code? Falsification-driven verification and testing. *Autom Softw Eng* **25**, 917–960 (2018). https://doi.org/10.1007/s10515-018-0240-y

実際、この方法って上手く使えるの...?

Rustで、Kaniとmutationツールを使って、実際に試してみた

アプローチ:

1. ChatGPTにQuickSortを書かせる（よくある実装が行われることを期待）
2. Kaniの検証用コードを記述
3. 正常に検証が終了することを確認
4. mutationを実行し、それぞれのファイルに対してKaniを実行
5. 実行結果から、Kaniの記述の甘さを指摘するようなケースがあるかを確認

実験の設定

以下の2つのmutationツールを使用し、比較を行った

1. cargo-mutantsを使用した場合
   生成されたmutantsの数：19個
   https://github.com/sourcefrog/cargo-mutants
2. universalmutatorを使用した場合
   生成されたmutantsの数：127個
   https://github.com/agroce/universalmutator

実験結果

mutationツールによって、実行結果にかなりの差が出る

- cargo-mutantsでは、falsificationに有益なmutationが生成されなかった
- universalmutatorでは、非自明な成功例は生成されたが、
  期待していたようなものはなかった
  (i.e. ソート後の配列の中身をランダムにするなど）

```rust
#[cfg(kani)]
mod verification {
    use crate::quicksort;
    #[kani::proof]
    #[kani::unwind(3)]
    fn verify_quicksort() {
        const SIZE: usize = 2;
        let mut a = vec![];
        let s = kani::any();
        kani::assume(s > 0 && s <= SIZE);
        for i in 0..s {
            let v = kani::any();
            println!("LOG: ref[{i}] = {v}");
            a.push(v);
        }

        let _ref = a.clone();

        let len = a.len();
        quicksort(&mut a, 0, len - 1);
        let mut prev = a[0];
        for i in 0..s {
            println!("LOG: a[{i}] = {}", a[i]);
            assert!(prev <= a[i]);
            prev = a[i];
        }

        // let v: i32 = kani::any();
        // let mut count = 0;
        // let mut qcount = 0;
        // for i in 0..s {
        //     if _ref[i] == v {
        //         count += 1;
        //     }
        //     if a[i] == v {
        //         qcount += 1;
        //     }
        // }

        // assert_eq!(count, qcount);
    }
}
```

```
===============

[ main.mutant.60.rs ]
--- ../../src/main.rs   2024-12-09 16:04:47
+++ main.mutant.60.rs   2024-12-09 16:17:12
@@ -5,7 +5,7 @@
         if p > 0 {
             quicksort(arr, low, p - 1); // Avoid underflow
         }
-        quicksort(arr, p + 1, high);
+        quicksort(arr, p + (1+1), high);
     }
 }

===============

[ main.mutant.61.rs ]
--- ../../src/main.rs   2024-12-09 16:04:47
+++ main.mutant.61.rs   2024-12-09 16:17:12
@@ -5,7 +5,7 @@
         if p > 0 {
             quicksort(arr, low, p - 1); // Avoid underflow
         }
-        quicksort(arr, p + 1, high);
+        quicksort(arr, p + (1-1), high);
     }
 }

===============

[ main.mutant.66.rs ]
--- ../../src/main.rs   2024-12-09 16:04:47
+++ main.mutant.66.rs   2024-12-09 16:17:12
@@ -5,7 +5,7 @@
         if p > 0 {
             quicksort(arr, low, p - 1); // Avoid underflow
         }
-        quicksort(arr, p + 1, high);
+        /*quicksort(arr, p + 1, high);*/
     }
 }

===============

[ main.mutant.92.rs ]
--- ../../src/main.rs   2024-12-09 16:04:47
+++ main.mutant.92.rs   2024-12-09 16:17:12
@@ -13,7 +13,7 @@
     let pivot = arr[high];
     let mut i = low;
     for j in low..high {
-        if arr[j] <= pivot {
+        if arr[j] < pivot {
             arr.swap(i, j);
             i += 1;
         }
===============
```

考察

- 最近のmutationツールが賢すぎて、「しっちゃかめっちゃか」な編集が行われない
  - 先行研究で使用したmutationツールは、Prologで実装した簡易的なツールで、コンパイルできないファイルも生成していた

    簡素なツールなので、簡単に自作はできそう

    - Replace an integer constant C by 0, 1, -1, ((C)+1), or ((C)-1).
    - Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator by another operator from the same class.
    - Negate the decision in an if or while statement.
    - Delete a statement.

    J. H. Andrews, L. C. Briand and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.

まとめ

- 検証を終えたら、一度、バグを埋め込んでみて、良い検証になっているかを確かめるのが王道的なやり方
- バグの埋め込みをmutationを使って行う手法も研究されている
- ただし、mutationツールの性能が良すぎると、うまくいかない
- 基本的には、手動でバグを埋め込みながら確かめて、その後にmutationツールを使って追加で確かめるのは良い方法かもしれない