EC4219: Software Engineering

Lecture 13 — Program Verification (4)

Invariant Inference by Guess-and-Check

Sunbeom So

2024 Spring

Hounini Overview

- Named after magician Harry Hounidi
- Originally proposed as annotation assistant for ESC/Java (extended static checker for Java).
- **Guess-and-Check**: Guess some annotations, and check if they are correct.
  - The annotations produced by Houdini are *sound* (inferred invariants are true invariants).
  - Generally applicable to inference of any types of invariants (loop invariants, function specifications, etc).
  - However, it is not complete. The synthesized annotations may not be sufficient to prove property.

Step 1: Guess Invariants

Many different techniques for guessing invariants.

- Collect candidates from source code based on heuristics
  - Expressions of the form $v_1 \ op \ v_2$ or $v_1 \ op \ c$, where v_1 and v_2 are variables used in source code and c is an "interesting" constant.
- Use dynamic analysis (Daikon approach)
  - Employ facts while running the program.
- All these techniques are heuristics in nature. Their effectiveness can differ depending on application domains.

Step 2: Check Invariants

- The checker only throws out candidate annotations that are refuted by the verifier.
- Loop invariant I is refuted if
  1. it is not implied by the loop precondition
  2. it is not preserved by the loop body
- Function precondition P is refuted if it does not hold at the function's call-site.
- Function postcondition Q is refuted if $\{P\}\ S\ \{Q\}$ is invalid, i.e., $P \rightarrow \textbf{pre}(Q, S)$ is invalid (S is the function body).

Pseudo Code of Houdini[1]

Algorithm 1 Houdini

Input: A program P to verify
Output: A conjunctive invariant A

```
A_0 ← enumerate speculated invariants
A ← A_0
while true do
    refuted ← Verify(P, A)
    if refuted = ∅ then
        return A
    A ← A \ refuted
```

- The algorithm returns the conjunctive invariant $I = \bigwedge_{b_i \in A} b_i$.
- **Termination**: Terminates after at most $|A_0|$ iterations.
- **Soundness**: Upon termination, annotations in A are true invariants.

[1]This algorithm assumes P has a sinlge loop.

Example: Finding Loop Invariants

Consider the simple code below.

```
1 i := 0;
2 j := -1;
3 while (i<1000) {
4   j := i;
5   i := i+1;
6 }
```

Suppose $A_0 = \{I_1 : i \geq 0, I_2 : i = j, I_3 : i < 1000, I_4 : i \leq 1000\}$. Compute the inductive invariant.

- The candidate I_2 is refuted because it is not implied by the precondition (the following is invalid):

$$\{true\}\ L1; L2\ \{I_2\}$$

- The candidate I_3 is also refuted because the following is invalid:

$$\{I_1 \wedge I_3 \wedge I_4 \wedge i < 1000\}\ L4; L5\ \{I_3\}$$

Property of Houdini Algorithm

Given a set of candidate loop invariants, Houdini finds the largest subset that is inductive (i.e., the strongest inductive invariant). Why? (proof by contradiction)

- Suppose Houdini returns the set $\boldsymbol{A}$, but there exists a stronger and inductive invariant $\boldsymbol{B}$, i.e., $\boldsymbol{B} \supset \boldsymbol{A}$ such that $\bigwedge_{b_i \in B} \boldsymbol{b_i}$ is inductive.
- This means that the algorithm must have eliminated some $\boldsymbol{b_i} \in \boldsymbol{B}$.
- This happens only when if either (1) $\boldsymbol{Pre} \rightarrow \boldsymbol{b_i}$ is invalid or (2) $\{I_B \wedge C\}\ Body\ \{b_i\}$ is invalid.
- But neither option is possible since $\boldsymbol{B}$ is inductive according to our assumption (contradiction!).

Houdini for Function Specifications

- Houdini is not limited to inferring loop invariants, and it can be used to infer function specifications.
- Suppose we have a candidate set of preconditions (P) and postconditions (Q), and initialize preconditions and postconditions of every function with P and Q, respectively.
- When analyzing a function F:
  - If verification fails due to the callee's precondition p, remove p from the callee's precondition set.
  - If verification fails because some postcondition q could not be established, remove q from the F's postcondition set Q.

Example: Finding Function Specifications

```
main ( ) { foo (5,0); }

foo (x, y) {
  if (x<=0) z := y; else z := bar(x,y);
  return z;
}

bar (x,y) { x := x-1; y := y+1; return foo(x,y); }
```

Suppose $P = \{P_1 : x \geq 0, P_2 : y \geq 0, P_3 : x = y, P_4 : x > 0\}$. Suppose also $Q = \{Q_1 : rv \geq 0, rv = 0\}$. Find function specifications for `foo` and `bar`.

Example: Finding Function Specifications (Cont'd)

```
main ( ) { foo (5,0); }
foo (x, y) {
  if (x<=0) z := y; else z := bar(x,y);
  return z;
}
bar (x,y) { x := x-1; y := y+1; return foo(x,y); }
```

- When analyzing `main`, we remove $P_3 : x = y$ for `foo`.
- When analyzing `foo`, we remove $P_3 : x = y$ for bar because `assert(x=y)` fails at bar's callsite.
- When analyzing `foo`, we remove $Q_2 : rv = 0$ for `foo` because `assert(rv=0)` fails ($rv = 5$).

Example: Finding Function Specifications (Cont'd)

```
main ( ) { foo (5,0); }
foo (x, y) {
  if (x<=0) z := y; else z := bar(x,y);
  return z;
}
bar (x,y) { x := x-1; y := y+1; return foo(x,y); }
```

- When analyzing `bar`, we remove $P_4 : x > 0$ for `foo`.
- When analyzing `bar`, we remove $Q_2 : rv = 0$ for `bar`.
- Iterate the same process, and nothing is refuted.
- The inferred function specification for `foo`:

$$P = \{x \geq 0 \land y \geq 0\} \text{ and } Q = \{rv \geq 0\}$$

- The inferred function specification for `bar`:

$$P = \{x \geq 0 \land y \geq 0\} \text{ and } Q = \{rv \geq 0\}$$

Summary

Houdini algorithm: a simple approach for automatically inferring (strongest) invariants:

- Pros: general applicability, easy to implement
- Cons: infer conjunctive invariants only, not property-directed (no guarantee that the inferred invariants are useful for verifying property)

Finding invariants still remains an active research area (probabilistic reasoning, domain-specific refinement, etc) – join if interested!