Software Reliability

Lecture 8

Invariant Generation Using Houdini

Alastair Donaldson
www.doc.ic.ac.uk/~afd

Static program verification depends on invariants

Procedure summarisation relies on pre- and post-conditions. These are **invariants**: the pre-condition must be **invariantly** true on method entry, the post-condition **invariantly** true on method exit

Loop summarisation uses a loop **invariant**: a fact that must be **invariantly** true when control reaches the loop head

Invariant generation is a challenging problem

We shall study **Houdini**, a simple method for static generation of invariants

Before we begin: recap on inductive invariants

```
i = 0;
x = 1;
while(i < 100) {
    i = i + 1;
    x = 1 - x;
}
```

`i >= 0` is a loop invariant

`x >= 0` is also a loop invariant

`i >= 0` is **inductive**: knowing only `i >= 0`, the loop body tells us `i >= 0` is maintained

```
assume(i >= 0);
if(i < 100) {
    i = i + 1;
    x = 1 - x;
    assert(i >= 0);
}
```

Assume only that `i` is non-negative (`i` could be 100000, `x` can be anything)

Based on just this info, `i >= 0` will still hold if we execute one more loop iteration

Before we begin: recap on inductive invariants

```
i = 0;
x = 1;
while(i < 100) {
    i = i + 1;
    x = 1 - x;
}
```

`i >= 0` is a loop invariant

`x >= 0` is also a loop invariant

`x >= 0` is **not inductive**: knowing only `x >= 0`, the loop body does not tells us `x >= 0` is maintained

```
assume(x >= 0);
if(i < 100) {
    i = i + 1;
    x = 1 - x;
    assert(x >= 0);
}
```

Assuming only that `x` is non-negative admits, for example, `x` being 100000 and `i` being 0

If `x` was 100000 and `i` was 0, `x` will be negative – the invariant is not maintained

Houdini in a nutshell

Input: a program **P**, and a set of **candidate** invariants

The candidate invariants are "guesses" at pre-conditions, post-conditions and loop invariants. Many of them will turn out to be wrong

Result: the unique **largest** subset of the candidates whose conjunction is an inductive invariant for the program

Worst case: this subset is **empty**

Best case: **all** candidates are shown to be invariants

Houdini in a nutshell

Where do the candidates come from?

It does not matter to Houdini: the program **and the candidates** are provided as input to Houdini

In practice, candidates could come from various sources, including:

- Cheap static analysis of source code
- Dynamic analysis (e.g. the *Daikon* method)
- Users (i.e., provided manually)

Some example uses of Houdini:

- Reducing false positives in ESC/Java (see recommended paper)
- Proving race-freedom of GPU kernels in GPUVerify (tool developed at Imperial)
- State-of-the-art device driver analysis (see Microsoft's Q tool)

Houdini for loop invariant generation: example

```
void foo() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;
  while(i < 10000)
      candidate i == 0,
      candidate i != 0,
      candidate i >= 0,
      candidate i > 0,
      candidate i < 10000,
      candidate i <= 10000,
      candidate x != y
  {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
  }
}
```

Using your intuition, which of these guesses are loop invariants?

Iteration 1: try to verify that all candidates are invariant

```
void foo_houdini_1() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;

  assert(i == 0); assert(i != 0); assert(i >= 0); assert(i > 0);
  assert(i < 10000); assert(i <= 10000); assert(x != y);

  havoc(temp, x, y, z, i);

  assume(i == 0); assume(i != 0); assume(i >= 0); assume(i > 0);
  assume(i < 10000); assume(i <= 10000); assume(x != y);

  if(i < 10000) {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;

    assert(i == 0); assert(i != 0); assert(i >= 0); assert(i > 0);
    assert(i < 10000); assert(i <= 10000); assert(x != y);

    assume(false);
  }
}
```

Two assertions can fail:
`assert(i != 0);`
`assert(i > 0);`

None of these assertions fail. Why?

Kill candidates `i != 0` and `i > 0`

```
void foo() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;
  while(i < 10000)
      candidate i == 0,
      candidate i != 0,
      candidate i >= 0,
      candidate i > 0,
      candidate i < 10000,
      candidate i <= 10000,
      candidate x != y
  {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
  }
}
```

Iteration 2: try to verify that remaining candidates are invariant

```
void foo_houdini_2() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;

  assert(i == 0); assert(i >= 0);
  assert(i < 10000); assert(i <= 10000); assert(x != y);

  havoc(temp, x, y, z, i);

  assume(i == 0); assume(i >= 0);
  assume(i < 10000); assume(i <= 10000); assume(x != y);

  if(i < 10000) {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;

    assert(i == 0); assert(i >= 0);
    assert(i < 10000); assert(i <= 10000); assert(x != y);

    assume(false);
  }
}
```

Two assertions can fail:
`assert(i == 0);`
`assert(x != y);`

What changed to allow these assertions to start failing?

Kill candidates `i == 0` and `x != y`

```
void foo() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;
  while(i < 10000)
      candidate i == 0,
      candidate i != 0,
      candidate i >= 0,
      candidate i > 0,
      candidate i < 10000,
      candidate i <= 10000,
      candidate x != y
    {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
  }
}
```

Iteration 3: try to verify that remaining candidates are invariant

```
void foo_houdini_3() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;

  assert(i >= 0);
  assert(i < 10000); assert(i <= 10000);

  havoc(temp, x, y, z, i);

  assume(i >= 0);
  assume(i < 10000); assume(i <= 10000);

  if(i < 10000) {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;

    assert(i >= 0);
    assert(i < 10000); assert(i <= 10000);

    assume(false);
  }
}
```

One assertion can fail: **`assert(i < 10000);`**

Kill candidate `i < 10000`

```
void foo() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;
  while(i < 10000)
      candidate i == 0,
      candidate i != 0,
      candidate i >= 0,
      candidate i > 0,
      candidate i < 10000,
      candidate i <= 10000,
      candidate x != y
  {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
  }
}
```

Iteration 4: try to verify that remaining candidates are invariant

```
void foo_houdini_4() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;

  assert(i >= 0);
  assert(i <= 10000);

  havoc(temp, x, y, z, i);

  assume(i >= 0);
  assume(i <= 10000);

  if(i < 10000) {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;

    assert(i >= 0);
    assert(i <= 10000);

    assume(false);
  }
}
```

Verification succeeds!

Result of Houdini

```
void foo() {
  int x = 1, y = 2, z = 3, temp;
  int i = 0;
  while(i < 10000)
      ~~candidate i == 0~~,
      ~~candidate i != 0~~,
      candidate i >= 0,
      ~~candidate i > 0~~,
      ~~candidate i < 10000~~,
      candidate i <= 10000,
      ~~candidate x != y~~
  {
    temp = x; x = y; y = z; z = temp;
    i = i + 1;
  }
}
```

Houdini tells us:
`(i >= 0 && i <= 10000)`
is an **inductive invariant** for the loop

Guarantee: this is the **strongest** inductive invariant for the loop that is a conjunction of these candidates

Observation: `x != y` is a loop invariant, but it is **not inductive** – knowing only `x != y`, the loop body does not guarantee that `x != y` is maintained

Houdini for loops: general case

Input:

- Procedure **P** containing:
  - loops with regular invariants
  - calls to procedures with summaries
  - assertions
- Set **C** of candidate invariants for the loops of **P**

Result:

- **P** is **CORRECT**, plus largest subset of **C** found to be an inductive invariant

or

- **P** may be **INCORRECT**: problem with an assertion, pre-condition or regular loop invariant

May be a false positive as this is static program verification

Houdini for loops: general case

```
enable each candidate invariant in P;
while(true) {
    result = StaticallyVerify(P); // Apply static program verification
    if(result == CORRECT) {
        return (CORRECT, candidates still enabled in P);
    } else if(result == INCORRECT due to failed candidate c) {
        disable c in P;
    } else {
        // Must have result == INCORRECT due to failed assertion,
        // or regular invariant in P
        return (INCORRECT, details of failure);
    }
}
```

Claims about Houdini

The procedure terminates:

- On each iteration, either:
  - (a) the program is verified (termination),
  - (b) a possible defect is reported (termination), or
  - (c) a candidate is eliminated
- There are only |**C**| candidates, so termination is guaranteed within |**C**| iterations

The procedure is sound:

- This is immediate because **StaticallyVerify** employs static program verification, which is sound

The computed invariant (in the case that **P is CORRECT**) **is the largest subset of C that is an inductive invariant:**

- Let’s see a proof-sketch of this

Proof sketch: Houdini computes **largest** inductive invariant

Suppose `I` **and** `J` are known to be an inductive invariants for a loop **while**(**c**) { **B** }. That is:

```
assert(I);
havoc(modset(B));
assume(I);
if(c) {
     B;
     assert(I);
     assume(false);
}
```

is correct

```
assert(J);
havoc(modset(B));
assume(J);
if(c) {
     B;
     assert(J);
     assume(false);
}
```

is correct

Then `I && J` must also be an inductive invariant for the loop. That is:

```
assert(I && J);
havoc(modset(B));
assume(I && J);
if(c) {
     B;
     assert(I && J);
     assume(false);
}
```

is correct

Proof sketch: Houdini computes largest inductive invariant

We have: `I` is inductive and `J` is inductive **=>** `I && J` is inductive

Consequence: For a set of candidates **C**, there is a unique largest subset { $\mathbf{d}_1$, …, $\mathbf{d}_n$ } of **C** such that $\mathbf{d}_1$ && … && $\mathbf{d}_n$ is inductive

Justification: if { $\mathbf{e}_1$, …, $\mathbf{e}_a$ } and { $\mathbf{f}_1$, …, $\mathbf{f}_b$ } are subsets with:

- $\mathbf{e}_1$ && … && $\mathbf{e}_a$ inductive and
- $\mathbf{f}_1$ && … && $\mathbf{f}_b$ inductive

then

- $\mathbf{e}_1$ && … && $\mathbf{e}_a$ && $\mathbf{f}_1$ && … $\mathbf{f}_b$ is also inductive (by the above)

Get the **largest** inductive set by merging all inductive sets

So, the unique largest inductive invariant exists

Remains to show why Houdini is guaranteed to compute it

Proof sketch: Houdini computes largest inductive invariant

Suppose `I` is known to be an inductive invariant for a loop **while**(**c**) { **B** }. That is:

```
assert(I);
havoc(modset(B));
assume(I);
if(c) {
    B;
    assert(I);
    assume(false);
}
```

is correct

If we **strengthen** `I` by conjoining some extra stuff, `X`, to it, `I && X` might not be an inductive invariant:

```
assert(I); assert(X);
havoc(modset(B));
assume(I); assume(X);
if(c) {
    B;
    assert(I); assert(X);
    assume(false);
}
```

might not be correct

Proof sketch: Houdini computes **largest** inductive invariant

Suppose we have:

P

```
assert(I);
havoc(modset(B));
assume(I);
if(c) {
    B;
    assert(I);
    assume(false);
}
```

CORRECT

but:

Q

```
assert(I); assert(X);
havoc(modset(B));
assume(I); assume(X);
if(c) {
    B;
    assert(I); assert(X);
    assume(false);
}
```

INCORRECT

It is not possible for `assert(I)` to fail in **Q**, because otherwise `assert(I)` would also fail in **P**

Consequence: Houdini will never kill a candidate **c** if **c** belongs to an inductive subset of candidates

Proof sketch: Houdini computes largest inductive invariant

Set of candidates **C** can be **implicitly** partitioned into **D** and **E**

- **D** is the largest inductive subset
- **E** is the rest

Of course, we **don't know** what **D** and **E** are before we run Houdini, but the sets exist

Houdini will successively kill elements of **E**, but will **never** kill elements of **D**

Eventually, only **D** will remain and it will be shown to be inductive

Limitations of Houdini approach

Only **conjunctive** invariants can be computed:

With candidates **`a`**, **`b`**, **`c`**, **`d`**:

- We may compute a conjunctive invariant such as **`a && b && d`**
- We will not compute an invariant that involves disjunction or negation, such as **`a || !b`**

With candidate set **C**, we can only compute an invariant over **C**: quality of invariant depends on good guesses

To get a high quality invariant, we should guess **aggressively**

But then many guesses will be **wrong**, and refuting bad candidates is expensive (requires invoking an SMT solver)

Houdini for procedures

Suppose procedures $\mathbf{P}_1, \ldots, \mathbf{P}_n$ have:

- candidate **loop invariants**
- candidate **preconditions**
- candidate **postconditions**

We can extend Houdini to find the largest subset of these candidates that is an **inductive invariant**

Inductive: using just these invariants, we can prove all the procedures correct and re-establish all the invariants

Loop invariants, preconditions, postconditions are all invariants in the general sense

Houdini for procedures: basic idea

Try to verify each procedure in turn

Possible error: could be a false positive

If verification fails due to a **non-candidate** precondition, postcondition or loop invariant, report **INCORRECT**

If verification of **foo** fails due to:

- **candidate postcondition** of **foo**, remove **foo**'s candidate postcondition
- **candidate loop invariant** in **foo**, remove **foo**'s candidate loop invariant
- **candidate precondition** of **bar** (because **foo** calls **bar**) remove **bar**'s candidate precondition

If all procedures verify with no candidate failures, report **CORRECT**

Otherwise repeat the process (re-verify everything)

Worked example

c_requires: shorthand for **candidate_requires**

c_ensures: similar

```
int x; int y;

void foo()
    c_requires y == 2*x, c_requires (x % 2) == 0, c_ensures  x >= 0,
    c_ensures  y == 2*x, c_ensures  (y % 2) == 0 {
 if(x < 1000) {
    x = x + 1;
    y = y + 2;
    bar();
  }
}

void bar()
    c_requires y == 2*x, c_requires (x % 2) == 0, ensures  y == 2*x,
    c_ensures (y % 2) == 0 {
  if(x > 0) {
    x = x - 1;
    y = y - 2;
    foo();
  }
}
```

1) Verify foo with all candidates

```
assume y == 2*x;
assume (x % 2) == 0;
if(x < 1000) {
  x = x + 1;
  y = y + 2;
  // assert bar's preconditions
  assert y == 2*x;
  assert (x % 2) == 0;
  // havoc bar's modset
  havoc x;
  havoc y;
  // assume bar's postconditions
  assume y == 2*x;
  assume (y % 2) == 0;
}
assert x >= 0;
assert y == 2*x;
assert (y % 2) == 0;
```

Summary for **bar** using **bar**'s current candidates

INCORRECT: kill candidate precondition of **bar**

INCORRECT: kill candidate postcondition of **foo**

Remaining candidates

```
int x; int y;

void foo()
    c_requires y == 2*x, c_requires (x % 2) == 0, c_ensures  x >= 0,
    c_ensures  y == 2*x, c_ensures  (y % 2) == 0 {
 if(x < 1000) {
    x = x + 1;
    y = y + 2;
    bar();
  }
}

void bar()
    c_requires y == 2*x, c_requires (x % 2) == 0, ensures  y == 2*x,
    c_ensures (y % 2) == 0 {
  if(x > 0) {
    x = x - 1;
    y = y - 2;
    foo();
  }
}
```

2) Verify bar with remaining candidates

```
assume y == 2*x;
if(x > 0) {
    x = x - 1;
    y = y - 2;
    // assert foo's preconditions
    assert y == 2*x;
    assert (x % 2) == 0;
    // havoc foo's modset
    havoc x;
    havoc y;
    // assume foo's postconditions
    assume y == 2*x;
    assume (y % 2) == 0;
}
assert y == 2*x;
assert (y % 2) == 0;
```

INCORRECT: kill candidate precondition of **foo**

Summary for **foo** using **foo**'s current candidates

Remaining candidates

```
int x; int y;

void foo()
    c_requires y == 2*x, c_requires (x % 2) == 0, c_ensures  x >= 0,
    c_ensures  y == 2*x, c_ensures  (y % 2) == 0 {
 if(x < 1000) {
    x = x + 1;
    y = y + 2;
    bar();
  }
}

void bar()
    c_requires y == 2*x, c_requires (x % 2) == 0, ensures  y == 2*x,
    c_ensures (y % 2) == 0 {
  if(x > 0) {
    x = x - 1;
    y = y - 2;
    foo();
  }
}
```

3) Verify foo with remaining candidates

```
assume y == 2*x;
if(x < 1000) {
  x = x + 1;
  y = y + 2;
  // assert bar's preconditions
  assert y == 2*x;
  // havoc bar's modset
  havoc x;
  havoc y;
  // assume bar's postconditions
  assume y == 2*x;
  assume (y % 2) == 0;
}
assert y == 2*x;
assert (y % 2) == 0;
```

Summary for **bar** using **bar**'s current candidates

CORRECT

4) Verify bar with remaining candidates

```
assume y == 2*x;
if(x > 0) {
    x = x - 1;
    y = y - 2;
    // assert foo's preconditions
    assert y == 2*x;
    // havoc foo's modset
    havoc x;
    havoc y;
    // assume foo's postconditions
    assume y == 2*x;
    assume (y % 2) == 0;
}
assert y == 2*x;
assert (y % 2) == 0;
```

Summary for **foo** using **foo**'s current candidates

CORRECT

Result from worked example

foo and **bar** have been shown to satisfy these specs:

```
int x; int y;

void foo()
    requires y == 2*x, ensures  y == 2*x, ensures  (y % 2) == 0 {
 if(x < 1000) {
    x = x + 1;
    y = y + 2;
    bar();
  }
}

void bar()
    requires y == 2*x, ensures  y == 2*x, ensures (y % 2) == 0 {
  if(x > 0) {
    x = x - 1;
    y = y - 2;
    foo();
  }
}
```

This **non-candidate** postcondition was proven

Basic Houdini algorithm with procedures

```
enable each candidate invariant in P_1, …, P_n;
done = false;
while( ! done ) {
   done = true;
   for i in { 1, …, n } {
      result = StaticallyVerify(P_i); // Apply static prog. verification
      if(result == INCORRECT due to failed candidate c) {
         disable c in P_1, …, P_n;
         done = false;
      } else if(result == INCORRECT due to failed non-candidate) {
         // Problem with: assertion, regular loop invariant or regular
         // postcondition of P_i, or regular precondition of some P_j
         return (INCORRECT, details of failure);
      }
   }
}
return (CORRECT, candidates still enabled in P_1, …, P_n);
```

Optimisations

- Repeatedly check **foo** until **foo** **verifies** or non-candidate **failure** is reported
- If verification of **foo** leads to a candidate failure, re-verify **foo** with the reduced candidate set
- Avoid unnecessary re-verification. If verifying **foo** kills:
  - **precondition** of **bar**:

    mark **bar** for re-verification
  - **postcondition** of **foo**:

    mark all procedures that *directly* call **foo** for re-verification
  - **loop invariant** of **foo**:

    no need to re-verify other procedures
- There are opportunities for parallelising Houdini – think about them

A demo of Houdini in the Boogie verification framework

Example 1: A hand-written Boogie program

Example 2: A Boogie program generated by the GPUVerify tool

Live demo