CSE 503

Software Engineering

Winter 2021

Invariants and reasoning about programs

February 10, 2021

Recap: In-class exercise

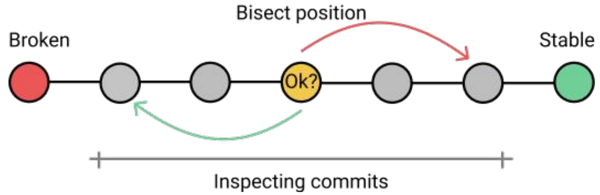
Git bisect: mostly binary search

- What’s the best, worst, and average case complexity of git bisect?

Recap: In-class exercise

Git bisect: mostly binary search

- What’s the best, worst, and average case complexity of git bisect?



Recap: In-class exercise

Git bisect: mostly binary search

- What’s the best, worst, and average case complexity of git bisect?

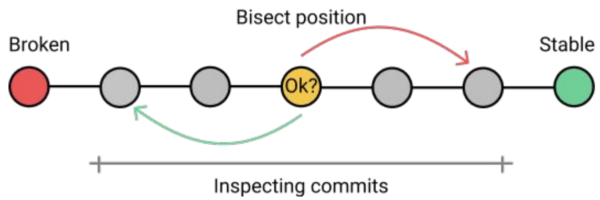
Undoing a commit vs. rewriting history

- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.

Recap: In-class exercise

Git bisect: mostly binary search
- What's the best, worst, and average case complexity of git bisect?

Undoing a commit vs. rewriting history
- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.
  - git revert
  - git reset
  - ...



Recap: In-class exercise

Git bisect: mostly binary search
- What's the best, worst, and average case complexity of git bisect?

Undoing a commit vs. rewriting history
- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.

DD: best case vs. worst case for duplicated inputs
- Given four inputs, which order is the best case vs. the worst case?

Recap: In-class exercise

Git bisect: mostly binary search
- What's the best, worst, and average case complexity of git bisect?

Undoing a commit vs. rewriting history
- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.

DD: best case vs. worst case for duplicated inputs
- Given four inputs, which order is the best case vs. the worst case?
  - 1123
  - 1213
  - 2311
  - 2113
  - ...

Course overview: the big picture

- **Week 1:** Introduction — **HW 1**
- **Week 2:** Abstract Interpretation
- **Week 3:** Abstract Interpretation — **HW 2**
- **Week 4:** Testing
- **Week 5:** Delta Debugging — **In-class exercise**
- **Week 6:** Invariants
- **Week 7:** Program Repair
- **Week 8:** Empirical Software Engineering
- **Week 9:** ML for Software Engineering
- **Week 10:** Wrap up — **Project presentation**

Course overview: the big picture

- **Week 1:** Introduction — **HW 1**
- **Week 2:** Abstract Interpretation
- **Week 3:** Abstract Interpretation — **HW 2**
- **Week 4:** Testing
- **Week 5:** Delta Debugging — **In-class exercise**
- **Week 6:** Invariants
- **Week 7:** Program Repair
- **Week 8:** Empirical Software Engineering
- **Week 9:** ML for Software Engineering
- **Week 10:** Wrap up — **Project presentation**

Let’s take a step back

Reasoning about programs

Use cases

- Verification/testing: ensure code is correct
- Prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - input array a is sorted
- Debugging: understand why code is incorrect

Reasoning about programs

Use cases

- Verification/testing: ensure code is correct
- Prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - input array a is sorted
- Debugging: understand why code is incorrect

Approaches

- Abstract interpretation
- Testing
- Delta debugging
- Slicing
- Theorem proving
- ...

Forward vs. backward reasoning

Forward reasoning

- Knowing a fact that is true before execution.
- Reasoning about **what must be true after execution**.
- Given a precondition, what postcondition(s) are true?

Backward reasoning

- Knowing a fact that is true after execution.
- Reasoning about **what must be true before execution**.
- Given a postcondition, what precondition(s) must hold?

What are the pros and cons for each approach?

Forward vs. backward reasoning

Forward reasoning

- More intuitive for most people
- Helps understand what will happen (simulates the code)
- Introduces facts that may be irrelevant to the goal
- Set of current facts may get large
- Takes longer to realize that the task is hopeless

Backward reasoning

- Usually more helpful
- Helps understand what should happen
- Given a specific goal, indicates how to achieve it
- Given an error, gives a test case that exposes it

Preconditions and postconditions

```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
    sum = sum + nums[i];
   else {
    sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Entry point

Exit point

What are preconditions and postconditions of this method (at the entry and exit points)?

Preconditions and postconditions

```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
    sum = sum + nums[i];
   else {
    sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Preconditions

- nums is not null
- `nums.length > 0`

Postconditions

- nums has not changed
- `n > 0`
- `sum >= 0`
- `return val >= 0`
- ...

(Loop) invariants

```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
     sum = sum + nums[i];
   else {
     sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Does this loop terminate?
What are preconditions,
postconditions,
and loop invariants?

(Loop) invariants

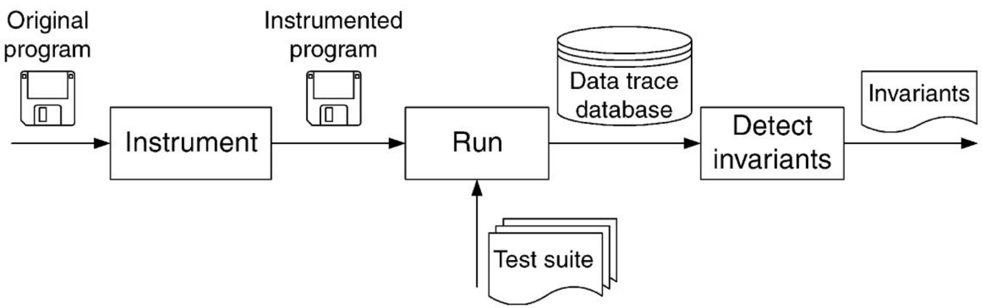
```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
     sum = sum + nums[i];
   else {
     sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Explicitly stating invariants is hard -- reasoning about inferred variants might be easier.

Daikon live example

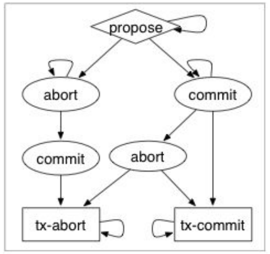
(https://plse.cs.washington.edu/daikon/download/doc/daikon/Example-usage.html#Detecting-invariants-in-Java-programs)

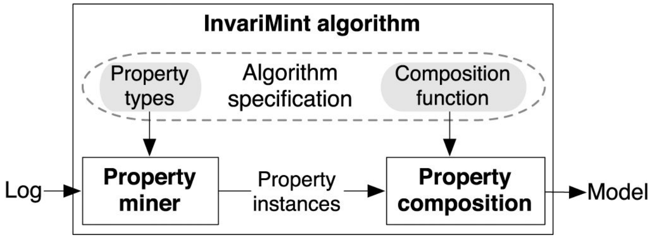
Daikon: general workflow



Daikon: other use cases

Synoptic





Daikon: discussion