Contract-based Program Repair without the Contracts

Yu Pei (yupei@polyu.edu.hk)

YOSER 2017, Beijing

Background

- Program faults: discrepancies between the specification and the implementation of programs
  - Detect: Testing
  - Diagnose: Fault localization
  - Correct: Fixing
    - Expensive and overwhelming to do manually
    - Can be automated in many cases

- Automated Program Repair (APR)
  - Input: A faulty program and a group of (unit) tests
    - Passing and failing
  - Output
    - A list of candidate fixes

Design-by-Contract

- Software systems consist of structured collections of cooperating software elements that cooperate on the basis of clear definitions of obligations and benefits
  - Contracts are assertions
    - Class invariants, routine pre- and postconditions
    - Eiffel, Java (JML), .NET (Code Contracts), etc.

```
class STACK[E]
  push(element: E)
      -- Push ‘element’ onto stack.
    require
      element /= Void
    ensure
      count = old count + 1
      top.equals(element)
```

    - Easy to understand and write, powerful, and executable
    - Faults trigger contract violations at runtime

Contract-based Automated Program Repair

- AutoFix
  - Automated testing + APR + IDE integration (for programs in Eiffel)
  - Use of contracts for fault localization and fix validation
  - Capability to propose high quality fixes

- Can we still generalize some of the techniques used for contract-based program repair to work effectively without user-written contracts?
  - JAID: automated program repair for Java

An Example Fault

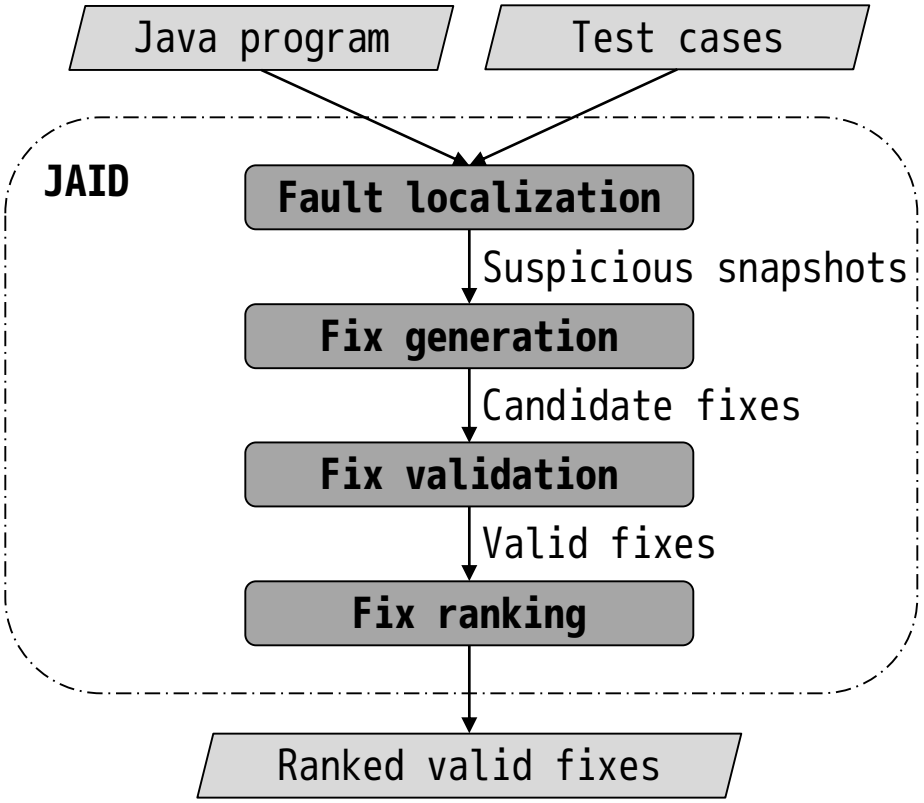
```
public static String abbreviate(String str, int lower, int upper, String appendToEnd){
  if (str == null) { return null; }          // abbreviate("Hello World!", 3, 8, "...")
  if (str.length() == 0) { return StringUtils.EMPTY; }                // ==> "Hello..."
  if (upper == -1 || upper > str.length()) { upper = str.length(); }
  if (upper < lower) { upper = lower; }
  StringBuffer result = new StringBuffer();
  int index = StringUtils.indexOf(str, " ", lower);
  if (index == -1) {
    // !! throws IndexOutOfBoundsException if lower > str.length()
    result.append(str.substring(0, upper));
    if (upper != str.length()) {
      result.append(StringUtils.defaultString(appendToEnd));
    }
  } else if (index > upper) {
    result.append(str.substring(0, upper));
    result.append(StringUtils.defaultString(appendToEnd));
  } else {
    result.append(str.substring(0, index));
    result.append(StringUtils.defaultString(appendToEnd));
  }
  return result.toString();
}
```

```
// Programmer-written fix
if(lower > str.length()){
  lower = str.length();
}
```

```
// fix suggested by JAID
if(lower >= str.length()){
  lower = str.length();
}
```

Contract-based APR without the Contracts

- Overall process



- Key feature
  - Building a rich abstraction of object state to guide fault localization and fix generation

Abstraction of Execution Traces

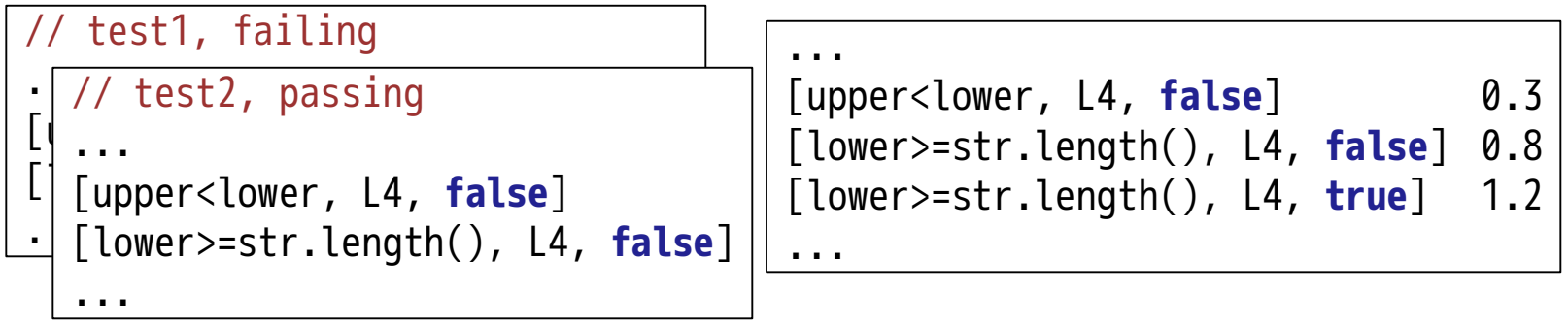
- State snapshots: **`[snapshot-expression, location, value]`**
  - Expressions to monitor (ETM) from method-to-fix
    - Purity analysis
      - Impure operators: `=`, `+=`, `-=`, `*=`, `/=`, `++`, `--`, and `new`
      - Unclear operators: method invocation `()`
      - Pure operators: the rest
  - Boolean-typed snapshot expressions
    - ETM^B, $!ETM^B$, ETM^B ✧ ETM^B, ETM^R ✦ ETM^R, ETM^N ✧ ETM^N
- Execution traces as sequences of state snapshots
  - Evaluate each snapshot expression at each program location
  - One sequence of state snapshots from each test execution

```
// test1, failing
...
[upper<lower, L4, false]
[lower>=str.length(), L4, true]
...
```

```
// test2, passing
...
[upper<lower, L4, false]
[lower>=str.length(), L4, false]
...
```

Fault Localization

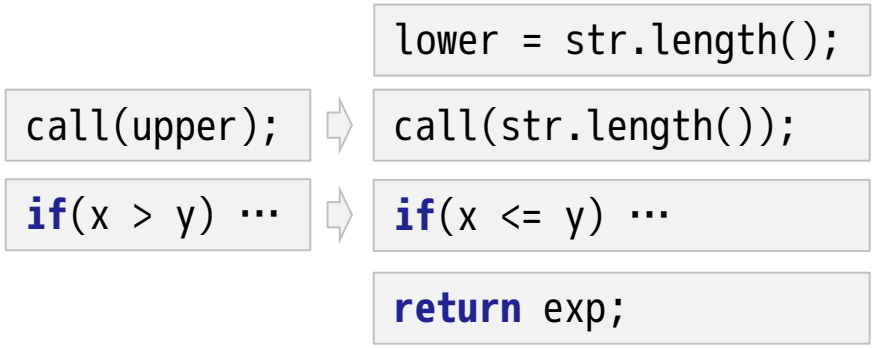
- Compute suspiciousness scores of the snapshots using two metrics
  - Frequencies of a snapshot observed in passing and failing test executions
  - Similarity between a snapshot expression and the code nearby its location



- Consider the most suspicious snapshots as potential fault causes

Fix Generation

- Construct fix actions to change the snapshot states
  - Modifying the state
  - Modifying an expression
  - Mutating a statement
  - Modifying the control flow



- Instantiate candidate fixes from schemas using fix actions and suspicious snapshots

```
action;
oldStatement;
```

```
// oldStatement;
action;
```

```
if(suspicious){
  action;
}
oldStatement;
```

```
if(!suspicious){
  oldStatement;
}
```

```
if(suspicious){
  action;
else{
  oldStatement;
}
```

```
// all candidate fixes
...
lower = str.length();
if(upper<lower){upper=lower;}
if(lower >= str.length()){
  lower = str.length();
}
if(upper<lower){upper=lower;}
...
```

Fix Validation and Ranking

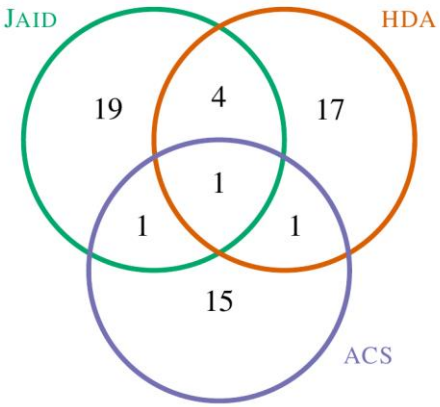
- Fix validation
  - Fixes that can make all the input tests pass are called valid

```
class ClassToFix {
  U methodToFix(T1 a1, T2 a2, ...) throws ASpecificException {
    switch (Session.getActiveFixId()) { // read from commond-line
      case 0: return methodToFix_0(a1, a2, ...); // call faulty method
      case 1: return methodToFix_1(a1, a2, ...); // call fix candidate 1
      ...
      case n: return methodToFix_n(a1, a2, ...); // call fix candidate n
      default: throw new IllegalStateException();
     }
   }
}
```

- Fix ranking
  - The more suspicious the related state snapshot, the higher the fix is ranked
  - Fixes derived from the same snapshot are ranked in order of generation

Experimental Evaluation of JAID

- Subjects
  - 138 faults from the Defects4J benchmark set of faults
- Effectiveness
  - Valid fixes to 29 faults, correct fixes to 25
- Performance
  - Average fixing time per bug is 90 minutes (median)
- Comparison with APR tools for Java programs
  - jGenProg, jKali, Nopol, xPar, HDA, ACS



Summary

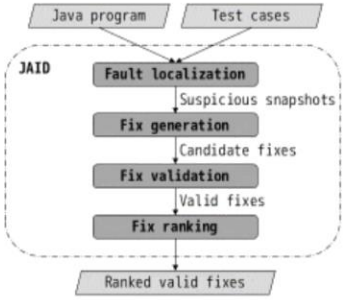
Motivation

- Contract-based automated program repair (AutoFix)
  - Automated testing and fixing for Eiffel programs
  - Capability to propose high quality fixes
  - Use of contracts for fault localization and fix validation
- Can we still generalize some of the techniques used for contract-based program repair to work effectively without user-written contracts?
  - JAID: automated program repair for Java

3

Contract-based APR without the Contracts

- Overall process



- Key feature
  - Building a rich abstraction of object state to guide fault localization and fix generation

5

Experimental Evaluation of JAID

- Subjects
  - 138 faults from the Defects4J benchmark set of faults
- Effectiveness
  - Valid fixes to 29 faults, proper fixes to 25
- Performance
  - Average fixing time per bug is 90 minutes (median)
- Comparison with others:
  - jGenProg, jKali, Nopol, xPar, HDA, ACS

10

Thank you!

Yu Pei (yupei@polyu.edu.hk)
Department of Computing
The Hong Kong Polytechnic University