Program Adjustment

Josh Branchaud and Eric Rizzi

CSE 990 - 4/9/13

State of the Art

1) Semantics to define behavior of program

2) Ability to find and locate bugs automatically

3) Ability to synthesize programs

Competent Programmer Hypothesis

- Brute Forcing a complete program is not an option
- Competent Programmer Hypothesis states that a competent programmer will write code that is almost correct.
- Rely instead on framework provided by programmer

Overview

- Project Overview
- Current Work
- Fault Localization
- Synthesis
- Concluding Remarks
- Questions and Answers

Project Overview
Concept

- Whole Project can be broken down into 4 individual parts
  - Identify line(s) causing error
  - Lower preconditions and assertions above
  - Raise postconditions and assertions below
  - Synthesize solution

Project Overview

A Quick Example

```
pre: (x != -1)
foo(x, y):
    a = x * y
    if(a <= 0):
        if(y < 0):
            y = y * -1
        else:
            if( x < 0):
                x = x * 1      //bug
    if(x < 0):
        x = x * -1
        y = y * -1
    ret = x + y
post: (ret >= 0)
```

Project Overview
Problems with Approach

- Multiple bugs
- Dependent bugs
- Synthesis problems
- Insufficient specifications

Project Overview

Existing Approaches

- Modular and Verified Automatic Program Repair
  - Francesco Logozzo and Thomas Ball
- SemFix: Program Repair via Semantic Analysis
  - Nguyen et al.

Goals of Logozzo

- Real time solutions to warnings raised by Checker
- Target and solve specific classes of bugs
- Use abstraction to handle even changes in control flow

Contract Repair

```
int [] ContractRepair(int index)
{
  var length = GetALength();
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
}
```

Logozzo: Step 1

- Identify areas of potential errors and generate traces
  - Floating point and Arithmetic Overflow
  - Off by One
  - Initializations
  - Contracts
  - Guards
  - Implicit and Explicit Constraints

Traces

- Individual runs through a program
  - Concrete
  - Abstract
- Includes information important to execution of program
  - Assertions
  - Assignments
  - Contracts

Traces According to Logozzo

Traces are a sequence of states

$$\vec{s} = \vec{s}_0 \ldots \vec{s}_{n-1}$$

The set of all finite Bad Traces is

$$\vec{B} = \left\{ s \in \vec{\Sigma}^{+} \mid \exists i \in \left[0, \left|\vec{s}\right|\right). s_i \in B \right\}$$

Definition of Repair

- Reduces the number of bad traces and increases the number of good traces.
- Two Types of Repairs
  - Verified Repair (Strong)
  - Verified Assertion Repair (Weak)

Logozzo: Step 2

- Repair Program by Applying Set Pattern
  - Backwards Must Analysis
    - Contracts
    - Initialization
    - Guards
  - Forward May Analysis
    - Off By One
    - Arithmetic Overflow
    - Floating Point Comparison
- Evaluate to make sure repair improved performance of program

Backwards Must Analysis

```
int foo(x){
   string s = null;
   if(x>0){
      s = "Hello World";
   return s.Length;
}
```

Forwards May Analysis

```
int foo(x){
   var args = new object[x];
   args[x] = key;
   returns args;
}
```

Lessons from Logozzo

- Incremental repair preferable to perfect repair
- Can oftentimes be multiple solutions to single bug
- Traces key to identifying errors
- Traces key metric for identifying improvement
- Program repair is only as good as specs given
- Implicit Specs just as important as explicit
- Using Human as oracle easier than applying fix

Project Overview

Existing Approaches

SemFix: Program Repair via Semantic Analysis

Nguyen et al. ICSE'13

- Semantic-based Program Fixing
- Uses symbolic execution, constraint solving, and program synthesis
- Given a buggy program and a test suite:
  - isolate the fault
  - infer specifications on the faulty statement
  - synthesize a fix* that satisfies the specifications

*only single line RHS and branch condition fixes

Project Overview
Existing Approaches

Isolate the fault

- Uses Tarantula (Jones et al. ICSE'02)
  - Run test suite
  - Distinguish between passing and failing tests
  - Build a ranked list of suspicious program statements

Suspiciousness score:

$$susp(s) = \frac{failed(s)/totalfailed}{passed(s)/totalpassed + failed(s)/totalfailed}$$

Existing Approaches

Infer Specifications on the Candidate Statement

Given:

- Program P
- Test Suite T
- Statement s in P (that needs repair)

Generate a repair constraint C that satisfies the test suite and can be used to generate a repair for s.

Project Overview
Existing Approaches

Infer Specifications on the Candidate Statement

For each test t_i in T that exercises s:

- Execute the program concretely on t_i's inputs
- When s is encountered, create a symbolic value for it
- Symbolically execute the rest of the program after s

The conjunction of the path conditions corresponding to passing test cases make up the repair constraint C.

Project Overview
Existing Approaches

Synthesize a Fix for the Candidate Statement

Uses Component-based Program Synthesis

- Repair Constraint (encodes input-output pairs)
- Components (e.g. constants, +, -, >, <, etc.)
- Solves for a fix made up of some components* that satisfies the repair constraint

*varying sets of components are selected until a solution is found

Existing Approaches

TCAS Example

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

```
int is_upward_preferred(int inhibit, int up_sep,
     int down_sep) {
  int bias;
  if(inhibit)
    bias = down_sep; //fix: bias=up_sep+100
  else
    bias = up_sep;
  if (bias > down_sep)
    return 1;
  else
    return 0;
}
```

Project Overview

Existing Approaches

TCAS Example - fault localization

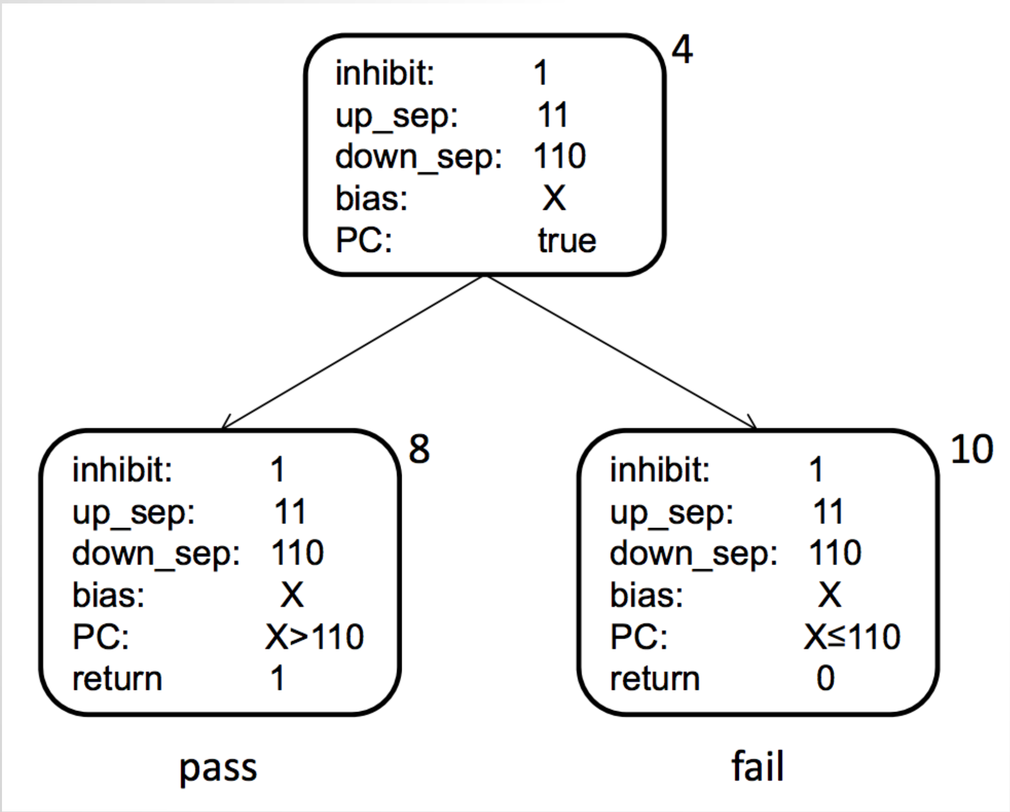
Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

Line	Score	Rank
4	0.75	1
10	0.6	2
3	0.5	3
7	0.5	3
6	0	5
8	0	5

Project Overview

Existing Approaches

TCAS Example - specification inference



Test 2: <1,11,110>
- f(1,11,110) > 110 & 1==1
- f(1,11,110) <= 110 & 0==1

Test 4: <1,-20,60>
- f(1,-20,60) > 60 & 1==1
- f(1,-20,60) <= 60 & 0==1

Test 1: <1,0,100>
- f(1,0,100) > 100 & 1==1
- f(1,0,100) <= 100 & 0==1

C: f(1,11,110) > 110 &
f(1,-20,60) > 60 &
f(1,0,100) > 100

Project Overview

Existing Approaches

TCAS Example - generate a repair

C: f(1,11,110) > 110 & f(1,-20,60) > 60 & f(1,0,100) > 100
Components: {variable, constant, "+", "-"}
Variables: {inhibit, up_sep, down_sep, ~~bias~~}

Progression:

- Constants only (c)
- "+" arithmetic expression (*var* + *c* or *var* + *var*)

f(inhibit,up_sep,down_sep) = up_sep + 100

Project Overview

Existing Approaches

TCAS Example

```c
int is_upward_preferred(int inhibit, int up_sep,
     int down_sep) {
  int bias;
  if(inhibit)
    bias = down_sep; //fix: bias=up_sep+100
  else
    bias = up_sep;
  if (bias > down_sep)
    return 1;
  else
    return 0;
}
```

Project Overview

Our Approach

The four main parts:

- Fault Localization
- Build Preconditions
- Build Postconditions
- Synthesize the Hole

Our Approach
Step 1: Fault Localization

- Use symbolic traces as basis
  - Generated using SPF listener
- SMT solver to partition between good and bad traces
- Satisfiable vs. Not Satisfiable
- Use similarities between bad traces and differences between good traces to localize bug
- Order lines according to likelihood of being source of bug

Our Approach
Step 2: Build Preconditions

- Entrance to bug composed of composition of all traces leading through line
  - Express possible state by ORing all traces
- SSA form important
  - Captures propagation of constraints

Project Overview

Running Example

```
pre: (x != 0)
foo(x, y):
    a = x * y
    if(a <= 0):
        if(y < 0):
            y = y * -1
        else:
            if( x < 0):
                x = x * 1
    if(x < 0){
        x = x * -1
        y = y * -1
    ret = x + y
post: (ret >= 0)
```

Our Approach

Step 3: Build Postconditions

- Exit of bug composed of all possible traces leaving buggy line
- Two possibilities to handle
  - Assignment Statement
  - Predicate of Conditional

Project Overview

Running Example

```
pre: (x != 0)
foo(x, y):
	a = x * y
	if(a <= 0):
		if(y < 0):
			y = y * -1
		else:
			if( x < 0):
				x = x * 1
	if(x < 0){
		x = x * -1
		y = y * -1
	ret = x + y
post: (ret >= 0)
```

Our Approach

Step 4: Synthesize Solution

- Previous steps reduced size of problem
- Have all possible information surrounding hole.
- Synthesize via escalating complexity
- Assume lines are of form
  - x = __________
  - if(x _____){
- Implication to pair related traces

Our Approach
Evaluation and Expectations

Evaluation with control systems:

- Traffic Collision Avoidance System (TCAS)
- Wheel Brake System (WBS)
- Elevator control system

Characteristics:

- Restricted to integers and booleans
- Lots of conditional control flow
- No looping
- 1 Step Mutations

Traces

A single path of execution through a program:

- Concrete
  - Start with some initial program state (inputs)
  - Execute a sequence of statements
- Symbolic
  - Start with symbolic program state (symbolic inputs)
  - Enumerate* paths through the program symbolically

* use CFG to guide SE toward a particular execution path

Fault Localization

Good vs. Bad Traces

Concrete:

- good - path exercised by a passing test case
- bad - path exercised by a failing test case

Symbolic:

- good - path that is satisfiable with the pre- and post-conditions
- bad - path that is unsatisfiable with the pre- and post-conditions

Fault Localization

Good vs. Bad Traces - Concrete

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

```
int is_upward_preferred(int inhibit, int up_sep,
     int down_sep) {
  int bias;
  if(inhibit)
    bias = down_sep; //fix: bias=up_sep+100
  else
    bias = up_sep;
  if (bias > down_sep)
    return 1;
  else
    return 0;
}
```

Fault Localization

Good vs. Bad Traces - Symbolic

	Trace1	Trace2	Trace3	Trace4	...
pre: (x != 0)					
foo(x, y):	foo(x,y):	foo(x,y):	foo(x,y):	foo(x,y):	
a = x * y	a = x * y	a = x * y	a = x * y	a = x * y	
if(a <= 0):	(a <= 0)	(a <= 0)	(a <= 0)	!(a <= 0)	
if(y < 0):	(y < 0)	!(y < 0)	!(y < 0)		
y = y * -1	y = y * -1				
else:		else:	else:		
if(x < 0):		!(x < 0)	(x < 0)		
x = x * 1			x = x * 1		
if(x < 0){			(x < 0)	(x < 0)	
x = x * -1			x = x * -1	x = x * -1	
y = y * -1			y = y * -1	y = y * -1	
ret = x + y	ret = x + y	ret = x + y	ret = x + y	ret = x + y	
post: (ret >= 0)					

Statistical Fault Isolation

Concrete:

$$susp(s) = \frac{failed(s)/totalfailed}{passed(s)/totalpassed + failed(s)/totalfailed}$$

- Tarantula collects traces by executing test suite
- Computes *susp(s)* for all statements

Symbolic:

- Use JPF/SPF to collect symbolic traces
- Check SAT of traces combined with specifications
- Compute *susp(s)* for all statements

Line	Score	Rank
4	0.75	1
10	0.6	2
3	0.5	3
7	0.5	3
6	0	5
8	0	5

Synthesis

- Implication to match input with output
- Trying to create a function P which satisfies the following constraint

$$pre(x,\ldots) \Rightarrow P(x\ldots) \wedge post(x\ldots)$$

Super Easy Synthesis

```
pre: (z == 13) && (x == 13) && (y == 10)
foo(z, x, y){
   if(z == x){
      ret = 1;
   }else{
      ret = 0;
   }
   return ret;
}
post: (ret == 1)
```

Super Easy Synthesis

```
pre: (z == 13) && (x == 13) && (y == 10)
foo(z, x, y){
   if(z == x){
      ret = 1;
   }else{
      ret = 0;
   }
   return ret;
}
post: (ret == 1)
```

```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const h Int)
(assert (= z 13))
(assert (= x 13))
(assert (= y 10))

(define-fun hole ((h Int) (a Int) (b Int)) Int
  (ite (= h 0) a b)
)

(assert (= z (hole h x y)))
(check-sat)
(get-model)
```

Synthesis

- Hole of type if(hole){
- Create Simple Grammar
  - hole -> var op var
  - var -> variables in scope | true | false
  - op -> = | <= | < | > | >=
- Define rules for each element of grammar that map position in assignment to logical formula
  - Use SMT solver from there

Synthesis

- Hole of type if(hole){
- Create Simple Grammar
  - hole -> var op var
  - var -> variables in scope | true | false
  - op -> = | <= | < | > | >=
- Define rules for each element of grammar that map position in assignment to logical formula
  - Use SMT solver from there

Synthesis

- Define multiple grammars to fill in different types of holes
- Increase complexity of grammar to account for more complicated program statements.

TABLE III
THE CATEGORIZATION OF BASIC COMPONENTS

Level	Conditional Statement	Assign Statement
1	Constants	Constants
2	Comparison ($>, \geq, =, \neq$)	Arithmetic ($+, -$)
3	Logic ($\wedge, \vee$)	Comparison, Ite
4	Arithmetic ($+, -$)	Logic
5	Ite, Array Access	Array Access
6	Arithmetic ($*$)	Arithmetic ($*$)

Synthesis

- Takes concrete pairs representative of each symbolic path
  - Can generate inputs and outputs from SAT solver
- Iteratively updates constructed function until satisfies all constraints
- Can test same way would test any other repair

Concluding Remarks

Program Adjustment is possible by building off of existing approaches (Logozzo and SemFix) using:

- Statistical Fault Localization
- Localized Specification Construction
- Program Synthesis

References

James A. Jones, Mary Jean Harrold, and John Stasko. 2002. **Visualization of test information to assist fault localization**. In Proceedings of the 24th International Conference on Software Engineering (ICSE '02). ACM, New York, NY, USA, 467-477.

Hoang D. T. Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. **SemFix: Program Repair via Semantic Analysis**. In Proceedings of the 35th International Conference on Software Engineering (ICSE '13). ACM, San Francisco, CA, USA.

Jose, Manu, and Rupak Majumdar. 2011 **Cause clue clauses: error localization using maximum satisfiability**" ACM SIGPLAN Notices 46.6, 437-446.

Liblit, Ben, et al. 2003. **Bug isolation via remote program sampling**. ACM SIGPLAN Notices. Vol. 38. No. 5. ACM..

Comments and Questions