Constraint-Based Fault-Localization

Michel RUEHER

joined work with

Mohammed Bekkouche and Hélène Collavizza

University of Nice Sophia-Antipolis   I3S – CNRS

France

INFORMS Computing Society Conference 2015
Sunday, January 11th, 2015

Plan

Problem statement & Motivating example

Context: program verification / debugging

Input An imperative program with **numeric statements** (over integers or floating-point numbers)

An **assertion** to be checked

A **counterexample** that violates the assertion

Output **Information** on locations of potentially **faulty statements**

Fault-Localization – a major problem

- **Model checking**, testing

→ Generation of **counterexamples**:

  - Input data & wrong outputs (testing)
  - Input data & violated post condition / property

→ **Execution trace**

- **Problems**

  - Execution trace: often **lengthy** and **difficult** to understand
  - **Difficult to locate** the faulty statements

Debugging ⇒ difficult and time consuming

Fault-Localization – Key issues

- **What paths to analyse ?**
  - Path from the counterexample
  - **Deviations** from the path from the counterexample
- **How to identify the suspicious program statements**
  - Computing Maximal sets of statements satisfying the postcondition → *Maximal Satisfiable Subset*
  - *Computing Minimal sets of statements to withdraw → Minimal Correction Set ?*

Example

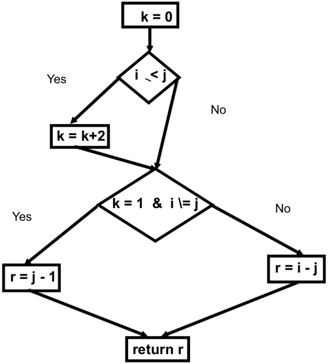
AbsMinus.java

```
 1 class AbsMinus {
 2     /* returns |i-j|, the absolute value of i minus j */
 3     /*@ requires     (i==0) && (j==1);
 4      @ ensures      (r==1);
 5      @*/
 6   int AbsMinusKO (int i, int j) {
 7     int r;
 8     int k = 0;
 9     if (i <= j) {
10       k = k+2; // error in assignement k = k+2 instead of k = k+1
11     }
12     if (k == 1 && i != j) {
13       r = j-i;
14     }
15     else {
16       r = i-j;
17     }
18     return r;
19   }
20 }
```

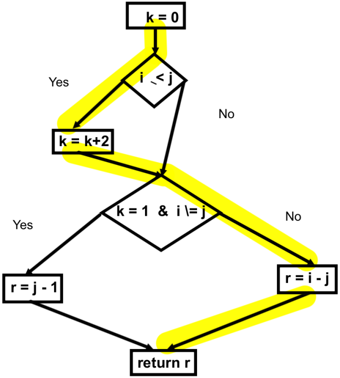
An error has been introduced in line 10

→ for the input data $\{i = 0, j = 1\}$, `r` $= -1$

Example (cont.)



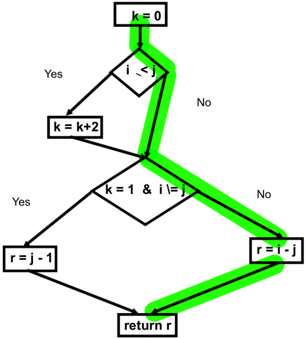
CFG of **AbsMinus**

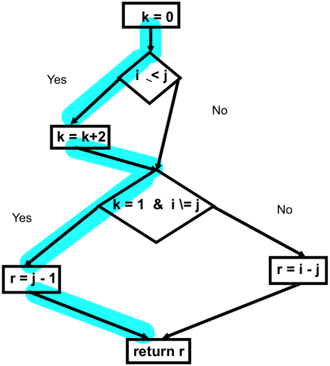


Faulty path for $\{i = 0, j = 1\}$

$\rightarrow$ suspicious statement: **{r = i - j}**

Example (cont.)





Change decision for 1st **IF**
Post-condition is **violated**
→ Path diversion **Rejected**

Change decision fort 2d **IF**
Post-condition **holds**
→ suspicious statements:
{cond. of 2d IF}, {k=0}, { k = k+2}

Proposed approach

- Explore the path of the counter-example and paths with **at most k deviations**
- Compute sets with **at most b_{mc} suspicious statements**

Bounds k and b_{mc} are mandatory
because there are an
exponential number
of paths and sets of suspicious statements

Formalization & Algorithms

Defining suspicious statements

Aim: Provide **helpful information** for error localization on numeric constraint systems:

- **MSS** Maximal Satisfiable Subset
  a generalization of MaxSAT / MaxFS considering maximality instead of maximum cardinality
  $M \subseteq C$ is a MSS $\Leftrightarrow$ M is SAT and $\forall c \in C \setminus M : M \cup \{c\}$ is UNSAT

- **MCS** Minimal Correction Set
  the complement of some MSS: removal yields a satisfiable MSS (it "corrects" the infeasibility)
  $M \subseteq C$ is a MCS $\Leftrightarrow$ $C \setminus M$ is SAT and $\forall c \in M : (C \setminus M) \cup \{c\}$ is UNSAT

Computing all MCS : CAMUS (Liffiton & Sakallah-2007

```
All_MCSes($\phi$)
1.  $\phi' \leftarrow$ AddYVars($\phi$)                     % Adds $y_i$ selector variables
2.  MCSes $\leftarrow \varnothing$
3.  k $\leftarrow$ 1
4.  while (SAT($\phi'$))
5.    $\phi'_k \leftarrow \phi' \wedge$ AtMost($\{\neg y_1, \neg y_2, . . . , \neg y_n\}$, k)
6.    while (newMCS $\leftarrow$ IncrementalSAT($\phi'_k$))       %All MCS of size K
7.          MCSes $\leftarrow$ MCSes $\cup$ {newMCS}
8.          $\phi'_k \leftarrow \phi'_k \wedge$ BlockingClause(newMCS)    % Excludes super sets for
                                                          % for size= k
9.          $\phi' \leftarrow \phi' \wedge$ BlockingClause(newMCS)       % Excludes super set
                                                          % for size > k
10. end while
11.   k$\leftarrow$k+1
12. end while
13. return MCSes
```

- *Incremental solver (MiniSAT) can be used in loop (l. 6) because constraints are only added but not external loop(l.4) since incrementing k relaxes constraints*
- *The set of yi variables assigned to false indicates the clauses in MCS*

LocFaults – Overall scheme

1. Building of the **CFG** of a program in DSA form
2. Translating the program and its specification in a set of **numerical constraints**
3. **Computing bounded MCS** of:
    - **C = CE ∪ PATH ∪ POST**
      CE: the counter-example
      PATH : constraints of the path of CE or of a diverted path
      POST: constraints of the post condition

    - **C = CE ∪ PATH' ∪ POST** where **PATH'** is a path with at most *k* **deviations** from the CE

→ **MCS on paths "closely" related to the CE**

LocFaults – Computing diverted paths

Process for $k = 1$

1. **Decision for 1st conditional statement** is switched and the input data of *CE* are propagated → new path **P'**
   Iff the ***CSP* of P' is satisfiable**, MCS are computed for **P'**
2. The process is restarted and **the decision of the next conditional statement of P is switched** (only one decision is changed on the whole path)

Process for $k > 1$

- A conditional node **n** is marked with the number of successful switches done on the current path before reaching **n**
- At step l, **decision for a node marked l' is only diverted iff $l' < l$**

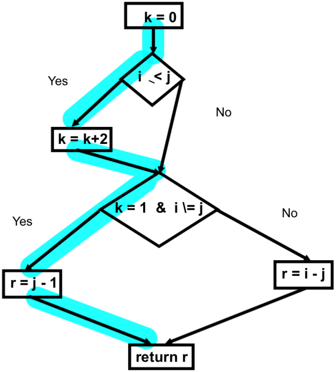
LocFaults – Computing MCS for diverted paths

- Let be :
  - **P**, a path generated by the propagation of **CE** and by **k decision switches** of conditional statements $\mathbf{cond_1}, .., \mathbf{cond_k}$
  - **C**, the constraints of **P**, and $\mathbf{C_k}$, the constraints generated by the assignments **occurring before** $\mathbf{cond_k}$ along $\mathbf{P_k}$

If **C** $\cup$ **POST holds**:

- $\{\neg\mathbf{cond_1}, .., \neg\mathbf{cond_k}\}$ is a **potential correction**,
- The **MCS** of $\mathbf{C_k} \cup \{\neg\mathbf{cond_1}, .., \neg\mathbf{cond_k}\}$ are **potential corrections**
  Note: $\{\neg cond_1, .., \neg cond_k\}$ is a "hard" constraint

Computing MCS for diverted paths – Example



CE: $\{i = 0, j = 1\}$
$\mathbf{cond_1} : \neg(\mathbf{k_1 = 1} \;\&\; \mathbf{i \neq j})$
$\mathbf{P_k}$: path in blue
$\mathbf{C_k} \cup \neg\mathbf{cond_1} : \mathbf{k_0 = 0} \wedge \mathbf{k_1 = k_0 + 2} \wedge$
$\neg((\mathbf{k_1 = 1} \;\&\; \mathbf{i \neq j}))$

Potential corrections:
$\{\mathbf{k_0 = 0}\}, \{\mathbf{k = k + 2}\}, \{\mathbf{k = 1} \;\&\; \mathbf{i \neq j}\}$

Experiments

Experiments - Systems and tools

- **LocFaults**:
  - → **CPBPV** (Constraint-Programming Framework for Bounded Program Verification) to generate the CFG and CE
  - → **CP** solver of **IBM ILOG CPLEX**
- **BugAssist** (Rupak Majumdar and Manu Jose):
  - → **CBMC**
  - → MaxSAT solver **MSUnCore2**

Experiments - Benchmarks

- **TCAS** :
  - **Aircraft collision avoidance system**
  - **173 lines of C code** with almost no arithmetic operations
  - The suite contains **41 faulty versions**
- **Tritype**

  Input: three **positive integers**, the triangle sides
  Output:
  - value 2 if the inputs correspond to an isosceles triangle
  - value 3 if the inputs correspond to an equilateral triangle
  - value 1 if the inputs correspond to a scalene triangle
  - value 4 otherwise.

Experiments - Results on TCAS suite

- **Computation times**: no significant difference
- At most **one deviation** required except for version `V41` (2 deviations required)
- Size of the set of suspicious instructions identified : in general larger for **BUGASSIST** than for **LOCFAULTS**
- **BUGASSIST** **identifies a bit more errors** than **LOCFAULTS**
- **LOCFAULTS** reports a **set of MCS for each faulty path**
  - → error localization process is much more easier than with the single set of suspicious errors reported by **BUGASSIST**

Experiments - Error on Tritype

- **TritypeV1** : error in the last assignment of the program
- **TritypeV2** : error in a nested condition, just before the last assignment
- **TritypeV3** : the error is an assignment and will entail a bad branching
- **TritypeV4**: error in condition, at the beginning of the program
- **TritypeV5** : two wrong conditions in this program
- **TritypeV6** : a variation that returns the *perimeter of the triangle*
- **TritypeV7** : a variation that returns the *product of the 3 sides*
- **TritypeV8** : a variation that computes the *square of the surface of the triangle by using Heron's formula*

Experiments - Results on `Tritype` (cont.)

P	CE	E	LocFaults				BugAssist
			0	1	2	3	
V1	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48},{30},{25}	{29, 32} {53, 57},{30}, {25}	/	{26, 27, 32, 33, 36, 48, 57, 68}
V2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{27},{25}	{29, 57} {32, 44}	/	{21, 26, 27, 29, 30, 32, 33, 35, 36, 33, 35, 36, 53, 68}
V3	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25}	{33, 45}	/	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
V4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45},{33},{25}	{26, 32}	{32, 35, 49} {32, 35, 53} {32, 35, 57}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
V5	$\{i = 2, j = 3, k = 3\}$	32, 45	{40}	{26} {29}	{32, 45} {35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
V6	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}

Suspicious statements on Tritype $V_1 - V_7$

Experiments - Results on Tritype (cont.)

P	CE	E	LocFaults		BugAssist
			0	1	
V7	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{27},{32}	$\{72, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62\}$
V8	$\{i = 3, j = 4, k = 3\}$	61	{61}	{29} {35},{30},{25}	$\{19, 61, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 54\}$

Suspicious statements on Tritype $V_8 – V_9$

Experiments - Results on Tritype (cont.)

Program	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
TritypeV7	$0,722s$	$0,051s$	$0,112s$	$0,119s$	$0,144s$	$0,140s$	$20,373s$
TritypeV8	$0,731s$	$0,08s$	$0,143s$	$0,156s$	$0,162s$	$0,216s$	$25,562s$

Computation times for non linear Trityp programs (V_8 – V_9)

Related Work & Conclusion

Related Work

- **BugAssist**:
    - \+ **Global approach** based on MaxSat
    - \- Merges the complement of MaxSat in a single set of suspicious statements
    - \- **Not efficient for programs with numeric statements**
- System based on **ranking of suspicious statements** (`Tarantula, Ochiai, AMPLE Debugging JUnit Tests in Eclipse, Jaccard,...`)
    - \+ Easy to implement
    - \- Require a huge number of test case and an **accurate Oracle**

Conclusion

- **Flow-based and incremental approach**
  - → **locates the errors around the path of the counter-example**
- **Constraint-based framework**
  - → well adapted for handling **arithmetic operations**
  - → can be extended in straightforward way for handling programs with **floating-point numbers computations**

Further Work: Improving constraint solving process

- **Adding redundant constraints**

  `res = s*(s-i)*(s-j);`

  → **res $\geq$ s, res $\geq$ (s-i), res $\geq$ (s-j)**

- **Combining symbolic simplification with CSP filtering techniques**

  `res = s*(s-i)*(s-j)*(s-i);`

  → identifying the square expression