Constraint-Based Error Localization

Mohammed, Bekkouche, Hélène Collavizza, Michel Rueher

University of Nice Sophia-Antipolis I3S – CNRS

France

Réunion VACSIM – 14 Octobre 2014

Plan

Problem: informal presentation

- **Model checking**, testing

  Generation of counterexamples

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

  $\rightarrow$ Execution trace

- **Problems**

  - Execution trace: often lengthy and difficult to understand
  - Location of the portions of code that contain errors

  $\rightarrow$ **Very expensive**

Goals

- Provide helpful information for error localization on numeric constraint systems
- Two categories of information
  - How much of an unsatisfiable constraint set can be satisfied ?
    $\rightarrow$ Minimal Correction Set (**MCS**)
    MaxSAT, Max CSP, MaxFS
  - Where in the constraint set the "problem" lies ?
    $\rightarrow$ Minimal Unsatisfiable Core (MUC),
    Irreducible Inconsistent Subsystems (IIS)

Definitions

- **MUS** Minimal Unsatisfiable Subset
  aka Irreducible Inconsistent Subsystem (IIS)
  $M \subseteq C$ is a MUS $\Leftrightarrow$ M is UNSAT and $\forall c \in M : M \setminus \{c\}$ is SAT

- **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(ϕ)

1. ϕ′ ← AddYVars(ϕ) *% Adds y_i selector variables*
2. MCSes ← ∅
3. k ← 1
4. **while** (SAT(ϕ′))
5. $\phi'_k \leftarrow \phi' \wedge$ **AtMost**($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$, k)
6. **while** (newMCS ← IncrementalSAT(ϕ'_k)) *%All MCS of size K*
7. MCSes ← MCSes ∪ {newMCS}
8. $\phi'_k \leftarrow \phi'_k \wedge$ **BlockingClause**(newMCS) % Excludes super sets for *% for size k*
9. ϕ′ ← ϕ′ ∧ **BlockingClause**(newMCS) *% Excludes super set % for size > k*
10. **end while**
11. k←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*

Computing all MCS – Example

- $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$
- $\phi = (x_1) \wedge (\neg x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$
- $\phi' = (\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3)$
- **K = 1**

  $\neg y_1 \wedge \neg x_1 \wedge (\neg x_1 \vee x_2) \vee \neg x_2 \vee (\neg x_1 \vee x_3) \wedge \neg x_3$: SAT $(\neg x_1 \wedge \neg x_2 \wedge x_3) \rightarrow$ **MCS : (C_1)**

  Adding : $\neg\neg y_1$, so $(\neg y_1 \vee x_1)$ reduces to x_1

  $x_1 \wedge \neg y_2 \wedge (\neg x_1 \vee x_2) \vee \neg x_2 \vee (\neg x_1 \vee x_3) \wedge \neg x_3$: UNSAT

  $x_1 \wedge \neg x_1 \wedge \neg y_3 \wedge \neg x_2$ ….: UNSAT

  …
- **K = 2**

  $\phi' = (\neg y_1 \vee x_1) \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3) \wedge \mathbf{y_1}$

  $= x_1 \wedge (\neg y_2 \vee \neg x_1) \wedge (\neg y_3 \vee \neg x_1 \vee x_2) \wedge (\neg y_4 \vee \neg x_2) \wedge (\neg y_5 \vee \neg x_1 \vee x_3) \wedge (\neg y_6 \vee \neg x_3)$

  $= x_1 \wedge \neg y_2 \wedge \neg y_3 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$:UNSAT

  $x_1 \wedge \neg x_1 \wedge \ldots$. :UNSAT

  …
- **K = 3**

  $x_1 \wedge \neg y_2 \wedge \neg y_3 \wedge \neg y_4 \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3)$:UNSAT

  $x_1 \wedge \neg y_2 \wedge (\neg x_1 \vee x_2) \wedge \neg y_4 \wedge \neg y_5 \wedge \neg x_3$: SAT $(x_1, \neg y_2, x_2, \neg y_4, \neg y_5, \neg x_3)$: → **MCS : (C_2, C_4, C_5)**

  …

Computing one MUS from a set of MCSes (Liffiton & Sakallah-2007

Irreducible hitting set

Consider MCSes ={{C_1,C_2,C_3},{C_2,C_4}} : {C_1,C_2} is a hitting set but is not Irreducible

PropagateChoice(MCSes, thisClause, thisMCS)
for each clause ∈ thisMCS
    for each testMCS ∈ MCSes
        if (clause ∈ testMCS) then testMCS ← testMCS - {clause}
for each testMCS ∈ MCSes
    if (thisClause ∈ testMCS) then MCSes ← MCSes - {testMCS}
MaintainNoSupersets(MCSes) *% removes any set in MCSes that is **now** a % superset of some other*

Consider {{C_1,C_2,C_3},{C_2,C_4},{C_1, C_3,C_4,C_5}}, after selection of $C_1$of first MCS:
1. Remove C_2 and C_3 in other MCS → {{C_4},{C_1,C_4,C_5}}
2. Remove MCS where C_1 occurs → {{C_4}}

LocFaults

→ **Computing MCS** on programs with numerical computations

- **Input :**
  - A faulty program: postcondition does not hold
  - A counter-example

- **Output :** A **small** set of suspicious statements

LocFaults : détails (1)

- **Process**

  1. Building of the **CFG** of a program in DSA form
  2. Translating of the program and its specification in a set of numerical constraints
  3. Computing**MCS** with the counter-example *CE*, constraints of the corresponding PATH and the postcondition POST
     Note : CSP $C = CE \cup PATH \cup POST$ is inconsistant

- **Key points** : **MCS on paths "closely" related to the CE**

  $\rightarrow$ Path of CE
  $\rightarrow$ Paths with at most k deviations from the CE

LocFaults : details (2)

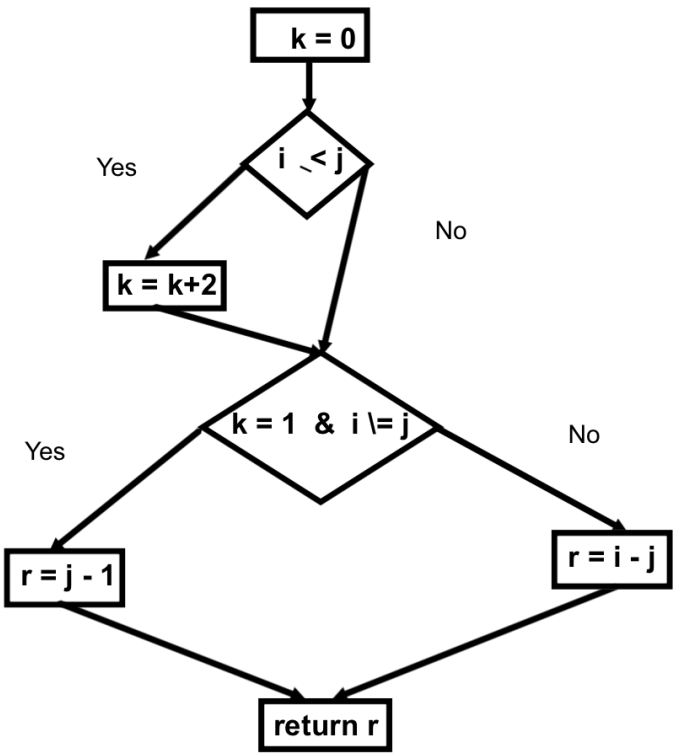
- Computing **bounded MCS** along the **path** of CE
- DFS Exploration of the CFG : propagation of CE and at most **k deviations** of conditional statements: $c_1, .., c_k$:
  - C: contraints along the path before c_k
  - IF $P \cup POST$ **holds**:
    - $\{\neg c_1, .., \neg c_k\}$ is a potential **correction**,
    - The **MCS** of $C \cup \{\neg c_1, .., \neg c_k\}$ are potential **corrections**

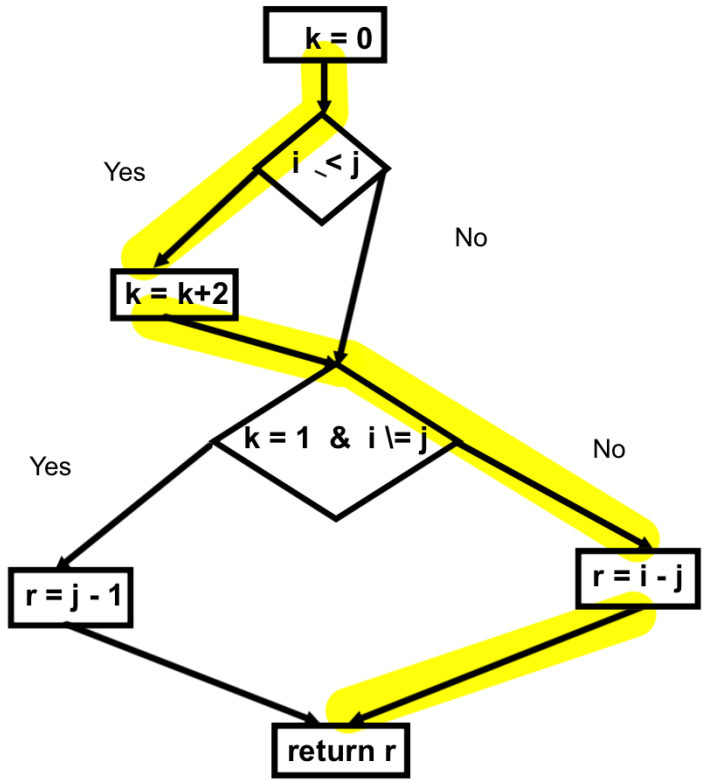
Example (1)

Program `AbsMinus` : an error has been introduced in line 10, thus for the input data $\{i = 0, j = 1\}$, it returns -1

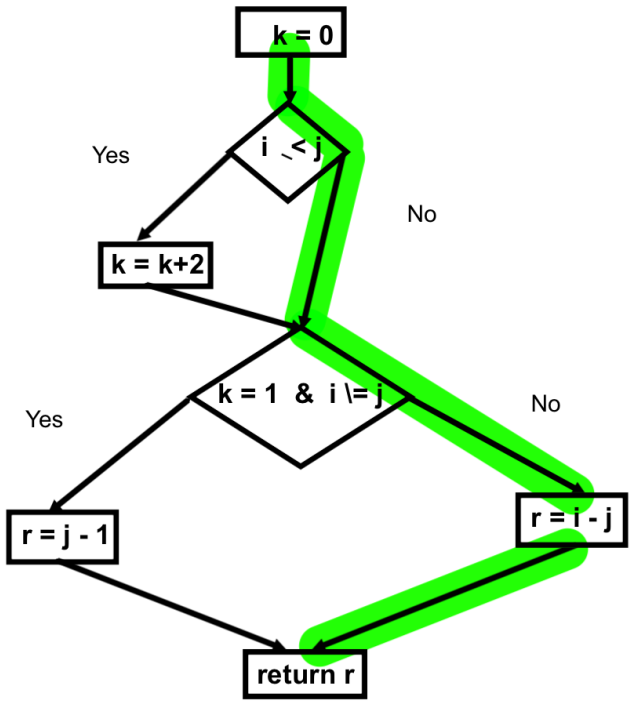
```
1 class AbsMinus {
2 /*returns |i−j|,the absolute value of i minus j*/
3 /*@ ensures
4  @ ((i < j) ==> (\result == j−i)) &&
5  @ ((i >= j) ==> (\result == i−j)); */
6  int AbsMinus (int i, int j) {
7    int result;
8    int k = 0;
9    if (i <= j) {
10        k = k+2; } // error : k = k+2 instead of k=k+1
11    if (k == 1 && i != j) {
12          result = j-i; }
13        else {
14          result = i-j; }
15 return result;
  }
}
```

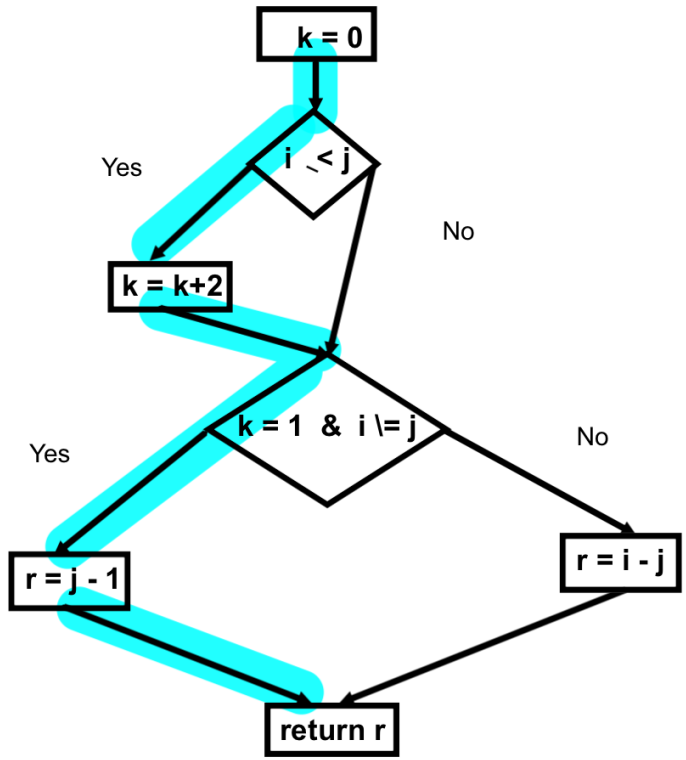
Example (2)


k = 0
i < j
Yes
No
k = k+2
k = 1 & i \= j
Yes
No
r = j - 1
r = i - j
return r


k = 0
i < j
Yes
No
k = k+2
k = 1 & i \= j
Yes
No
r = j - 1
r = i - j
return r

Example (3)


k = 0
i < j
Yes
No
k = k+2
k = 1 & i \= j
Yes
No
r = j - 1
r = i - j
return r


k = 0
i < j
Yes
No
k = k+2
k = 1 & i \= j
Yes
No
r = j - 1
r = i - j
return r

Experiments - Process

systems and tools

- **LocFaults**:

  → **MIP** solver of **IBM ILOG CPLEX**

  → **CPBPV** system to generate the CFG and CE

  → Benchmarks: **Java** programs

- **BugAssist**:

  → MaxSAT solver**MSUnCore2**

  → Benchmarks: ANSI-C programs

Experiments - Benchmarks

- **TCAS** : an aircraft collision avoidance system. The program contains 173 lines of C code with almost no arithmetic operations. The suite contains 41 faulty versions

- **Tritype** takes three positive integers as inputs (i, j, k) the triangle sides, and returns the value 2 if the inputs correspond to an isosceles triangle, the value 3 if they correspond to an equilateral triangle, the value 1 if they correspond to some other triangle, and the value 4 otherwise.

Experiments - Results on TCAS suite

- Computation times: no significant difference
- At most one deviation required except for version `v41` where two deviations were required
- Size of the set of suspicious instructions identified by BugAssistin general larger than the sum of the sizes of the sets of suspicious instructions generated by LocFaults
- BugAssistidentifies a bit more errors than LocFaults
- LocFaultsreports a **set of MCS for each faulty path** $\rightarrow$ the 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 statement of the program
- `TritypeV2` : error in a nested condition, just before the last assignment
- `TritypeV3` : the error an assignment and will entail a bad branching
- `TritypeV4`: error in condition, at the beginning of the program
- `TritypeV5` : wo wrong conditions in this program
- `TritypeV6` : a variation that returns the perimeter of the triangle
- `TritypeV7` : a variation that computes the square of the surface of the triangle by using Heron's formula

Experiments - Results on `Tritype` (2)

Program	Counter-example	Errors	LocFaults				BugAssist
			= 0	= 1	= 2	= 3	
TritypeV1	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48},{30},{25}	{29, 32} {53, 57},{30},{25}	/	{26, 27, 32, 33, 36, 48, 57, 68}
TritypeV2	$\{i = 2, j = 2, k = 4\}$	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}
TritypeV3	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25}	{33, 45}	/	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
TritypeV4	$\{i = 2, j = 3, k = 3\}$	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}
TritypeV5	$\{i = 2, j = 3, k = 3\}$	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}
TritypeV6	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}
TritypeV7	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{27},{32}	/	/	{72, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62}
TritypeV8	$\{i = 3, j = 4, k = 3\}$	61	{61}	{29} {35},{30},{25}	/	/	{19, 61, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 54}

Experiments - Results on `Tritype` (3)

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

Conclusion and Discussion

- **Flow-based and incremental** approach of LocFAults is a good way to help the programmer with bug hunting since it locates the errors around the path of t he counter-example

- **Constraint-based** framework is well adapted for handling arithmetic operations and it can be extended in straightforward way for handling programs with floating-point numbers computations