I3S Laboratory, UNS-CNRS

Constraint-Based Error Localization

LocFaults: A new flow-driven and constraint-based error localization approach

Mohammed Bekkouche, Michel Rueher, Hélène Collavizza

{bekkouche,helen,rueher}@unice.fr

I3S/CNRS, BP 121, 06903 Sophia Antipolis Cedex, France
University of Nice-Sophia Antipolis

SAC 2015
April 13 - 17, 2015

Outline

Introduction

Motive to solve the subject

Error localization is an important task to debug an erroneous program but complex at the same time

→ When a program is not conform to its specification, i.e., the program is erroneous :

- BMC(Bounded Model Checking) and testing tools can generate one or more counterexamples
- **The trace of the counterexample** is often long and complicated to understand
- The identification of erroneous portions of the code is hard even for experienced programmers

Introduction

The problem: inputs and goal

Inputs

- A program contradicts its specification
- The violated postcondition POST
- A counterexample CE provided by a BMC tool

Goal

A reduced set of suspicious statements allowing the programmer to understand the **origin of his mistakes**

Introduction

The ideas

1. The program is modeled in a CFG in DSA form
2. The program and its specification are translated in numerical constraints
3. *CE* : a counterexample, PATH : **an erroneous path**
4. The CSP $C = CE \cup PATH \cup POST$ **is inconsistent**

Key issues

- What are the erroneous instructions on *PATH* that make *C* **inconsistent** ?
- Which subsets remove to make *C* **feasible** ?
- What **paths** to explore ? → **path** of CE, **deviations** from CE

Example

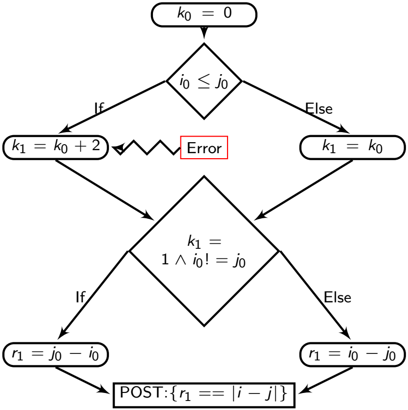
Calculate the absolute value of i-j

```
class AbsMinus {
/*returns|i-j|,the absolute value of i minus j*/
/*@ ensures
  @ (result==|i-j|);
  @*/
  void AbsMinus (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
       k = k+2; //error:k = k+2 instead of k=k+1
    }
    if (k == 1 && i != j) {
       result = j-i;
    }
    else {
       result = i-j;
    }
  }
}
```

Example

Calculate the absolute value of i-j

```
class AbsMinus {
/*returns|i-j|,the absolute value of i minus j*/
/*@ ensures
  @ (result==|i-j|);
  @*/
  void AbsMinus (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
       k = k+2; //error:k = k+2 instead of k=k+1
    }
    if (k == 1 && i != j) {
       result = j-i;
    }
    else {
       result = i-j;
    }
  }
}
```



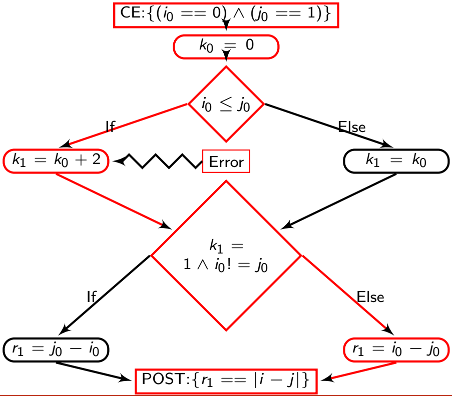
Example

The **path** of the counterexample

POST:$\{r_1 == |i - j|\}$

$\{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, r_1 = i_0 - j_0, r_1 = |i - j|\}$ **is inconsistent**

Only one MCS on the path : $\{r_1 = i_0 - j_0\}$



Exemple

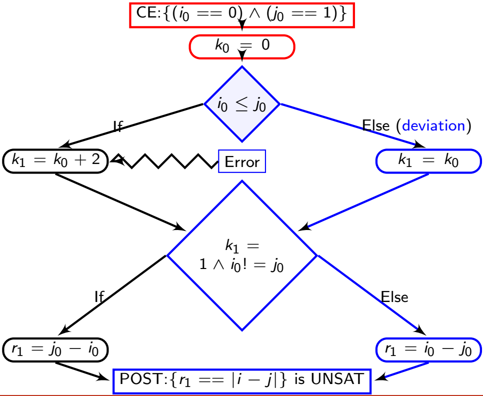
The **path** obtained **by deviating** the condition $i_0 \leq j_0$

The **deviated** condition : $\{i_0 \leq j_0\}$

$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = -1\}$

$P \cup \{r_1 = |i - j|\}$ **is inconsistent**

The deviation $\{i_0 \leq j_0\}$ **does not correct the program**



Example

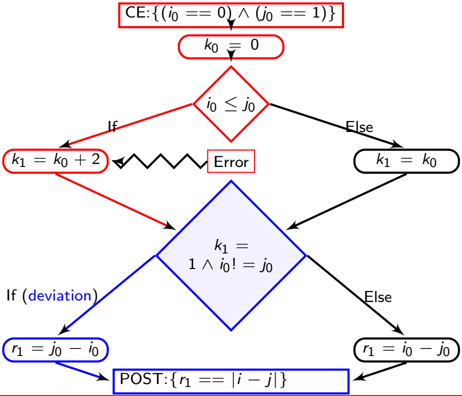
The **path by deviating** the condition $k_1 = 1 \wedge i_0 != j_0$

The **deviated** condition : $\{(k_1 = 1 \wedge i_0 != j_0)\}$

$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 2, r_1 = 1\}$

The deviation $\{(k_1 = 1 \wedge i_0 != j_0)\}$ **corrects** the program

$C = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, \neg(k_1 = 1 \wedge i_0 != j_0)\}$



Example

The **path by deviating** the condition $k_1 = 1 \wedge i_0 != j_0$

The **deviated** condition : $\{(k_1 = 1 \wedge i_0 != j_0)\}$

$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 2, r_1 = 1\}$

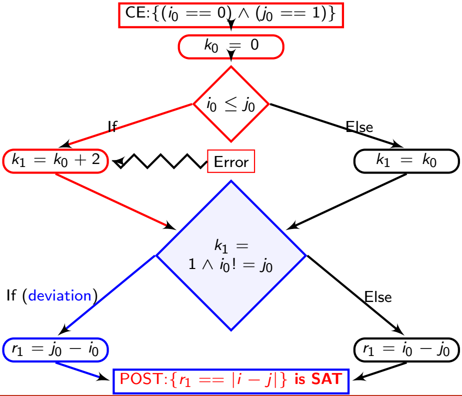
$P \cup \{r_1 = |i - j|\}$ **is consistent**

The deviation $\{(k_1 = 1 \wedge i_0 != j_0)\}$ **corrects** the program

$C = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, \neg(k_1 = 1 \wedge i_0 != j_0)\}$

C **is inconsistent**

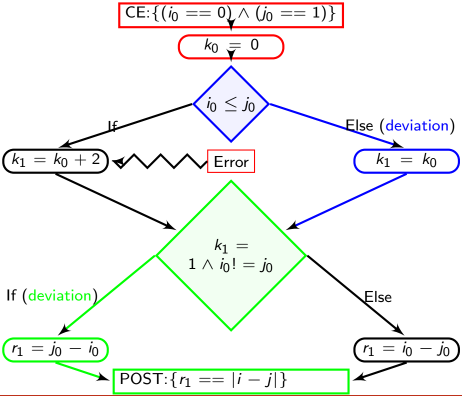
MCS on the path : $\{k_0 = 0\}$, $\{k_1 = k_0 + 2\}$



Example

The **path of a non-minimal deviation** : $\{i_0 \leq j_0, k_1 = 1 \wedge i_0! = j_0\}$

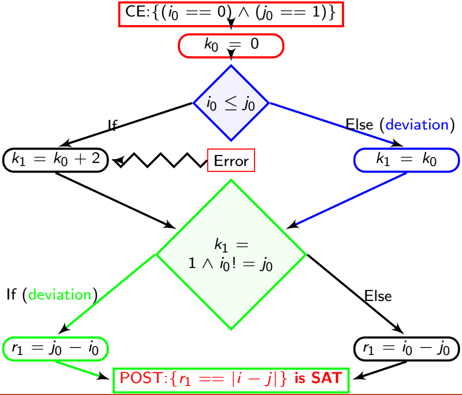
The **deviated** conditions :
$\{i_0 \leq j_0, (k_1 = 1 \wedge i_0! = j_0)\}$
$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\}$



Example

The **path of a non-minimal deviation** : $\{i_0 \leq j_0, k_1 = 1 \wedge i_0! = j_0\}$

The **deviated** conditions :
$\{i_0 \leq j_0, (k_1 = 1 \wedge i_0! = j_0)\}$
$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\}$
$P \cup \{r_1 = |i - j|\}$ **is consistent**
The deviation is not minimal



LocFaults approach

MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

$$M \subseteq C \text{ is a } \textbf{MCS} \Leftrightarrow \begin{cases} M \subseteq C \\ Sol(< X, C \backslash M, D >) \neq \emptyset \\ \nexists C'' \subset M : Sol(< X, C \backslash C'', D >) = \emptyset \end{cases}$$

LocFaults approach

MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

$$M \subseteq C \text{ is a } \mathbf{MCS} \Leftrightarrow \begin{cases} M \subseteq C \\ Sol(< X, C \backslash M, D >) \neq \emptyset \\ \nexists C'' \subset M : Sol(< X, C \backslash C'', D >) = \emptyset \end{cases}$$

LocFaults approach

MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

$$M \subseteq C \text{ is a } \textbf{MCS} \Leftrightarrow \begin{cases} M \subseteq C \\ Sol(< X, C \backslash M, D >) \neq \emptyset \\ \nexists C'' \subset M : Sol(< X, C \backslash C'', D >) = \emptyset \end{cases}$$

LocFaults approach

MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

$$M \subseteq C \text{ is a } \mathbf{MCS} \Leftrightarrow \begin{cases} M \subseteq C \\ Sol(< X, C \backslash M, D >) \neq \emptyset \\ \nexists C'' \subset M : Sol(< X, C \backslash C'', D >) = \emptyset \end{cases}$$

MCS: Example

- $C = \{c_1 : i = 0, c_2 : v = 5, c_3 : w = 6, c_4 : z = i + v + w, c_5 : ((z = 0 \vee i \neq 0) \wedge (v \geq 0) \wedge (w \geq 0))\}$ **is inconsistent**
- C has 4 **MCS**: $\{c_1\}$, $\{c_4\}$, $\{c_5\}$,$\{c_2, c_3\}$

LocFaults approach

(LocFaults) algorithm

- Isolation of **MCS** on the **path** of CE
- DFS exploration of CFG by propagating CE and **by deviating** at most k conditional statements $c_1, .., c_k$
  - P: propagation constraints derived from CE (of the form $variable = constant$)
  - C: constraints of **path** up to c_k
  - If $\underline{P \models POST}$:
    - * $\{\neg c_1, .., \neg c_k\}$ is a **correction**,
    - * **MCS** of $C \cup \{\neg c_1, .., \neg c_k\}$ are **corrections**
- A bound for the **MCS** calculated and the conditions **deviated**

Experimental evaluation
Tools used

- **LocFaults**: our implementation
  - → The IBM solvers CP OPTIMIZER and CPLEX
  - → The tool CPBPV to generate the CFG and CE
  - → Benchmarks: Java programs
- **BugAssist**: the tool of error localization for `BugAssist` approach
  - → The MaxSAT solver MSUnCore2
  - → Benchmarks: ANSI-C programs

Experimental evaluation

Programs built

- Variations on the **Tritype** program :
  → TritypeV1, TritypeV2,TritypeV3, TritypeV4, TritypeV5
  → TritypeV6 (returns **the perimeter** of the triangle)
  → TritypeV7, TritypeV8 (return **non linear expressions**)
- **TCAS(Traffic Collision Avoidance System)**, a realistic benchmark :
  → 1608 test cases, except cases for overflow *PositiveRAAltThresh* table
  → TcasKO . . . TcasKO41

Experimental evaluation

Results (**MCS** identified)

Program	Counter-example	Errors	LocFaults = 0	LocFaults = 1	LocFaults = 2	LocFaults = 3	BugAssist
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}

LocFaults provides a more informative and explanatory localization

Experimental evaluation

Results (computation times for non linear programs)

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

LocFaults is an order of magnitude faster than BugAssist on these two benchmarks

Experimental evaluation

Results (number of errors localized for TCAS)

Programme	Nb_E	Nb_CE	LF	BA
V1	1	131	131	131
V2	2	67	67	67
V3	1	23	23	13
V4	1	20	4	20
V5	1	10	9	10
V6	1	12	11	12
V7	1	36	36	36
V8	1	1	1	1
V9	1	7	7	7
V10	2	14	12	14
V11	2	14	12	14
V12	1	70	45	48
V13	1	4	4	4
V14	1	50	50	50
V16	1	70	70	70
V17	1	35	35	35
V18	1	29	28	29
V19	1	19	18	19
V20	1	18	18	18
V21	1	16	16	16
V22	1	11	11	11
V23	1	41	41	41
V24	1	7	7	7
V25	1	3	2	3
V26	1	11	7	11
V27	1	10	9	10
V28	1	75	74	58
V29	1	18	17	14
V30	1	57	57	57
V34	1	77	77	77
V35	1	75	74	58
V36	1	122	120	126
V37	1	94	21	94
V39	1	3	2	3
V40	2	122	72	122
V41	1	20	16	20

The performances of `LocFaults` and `BugAssist` are very similar on this programs well adapted for a Boolean solver

Related work

SAT-based approaches

BugAssist

- A BMC method, like ours
- **Major differences :**
  → It transforms the entire program into a SAT formula
  → It based on the use of MaxSAT solvers

+ Global approach
– The complement of the MaxSAT set does not necessarily correspond to the instructions on the same path
  → Displaying the union of these complements

Related work

Approaches based on systematic testing

Tarantula, Ochiai, AMPLE, Jaccard, Heuristics III

- Ranking of suspicious statements detected during the execution of a test battery

\+ Simple approaches

– Need **many** test cases
Approaches that require the existence of an oracle
→ Decide if the result of **tens of thousands** of test is **just**

Our framework is less demanding
→ Bounded Model Checking

Conclusion and future work

- Our flow-based and incremental approach is a good way to help the programmer with bug hunting
  $\rightarrow$ it locates the errors around the path of the counter-example
- We plan :
  - to develop an interactive version of our tool :
  $\rightarrow$ to provide the localizations one after the others
  $\rightarrow$ to take benefit from the user knowledge to select the condition that must be diverted
  - to extend our approach in straightforward way for error localization in programs with floating-point numbers computations