Error localization

A bounded constraint-based approach to aid for error localization

Bekkouche Mohammed, Collavizza Hélène, Rueher Michel

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271
06900 Sophia Antipolis, France
JFPC 2014

November 20, 2014

Plan

Introduction

Motivation

- BMC(Bounded Model Checking) and testing tools can generate one or more counterexamples
- A counterexample provides an execution trace
- **The trace of the counterexample** is often long and complicated to understand
- The identification of erroneous portions of the code is complex for the programmer

→ Need to develop localization tools **to assist the developer** in this task

Introduction

Motivation

- BMC(Bounded Model Checking) and testing tools can generate one or more counterexamples
- A counterexample provides an execution trace
- **The trace of the counterexample** is often long and complicated to understand
- The identification of erroneous portions of the code is complex for the programmer

→ Need to develop localization tools **to assist the developer** in this task

Introduction

The problem

Inputs

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

Outputs

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

Introduction

The problem

Inputs

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

Outputs

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 points

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

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 points

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

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 points

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

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 points

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

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**: $\{c_1\}$, $\{c_4\}$, $\{c_5\}$,$\{c_2, c_3\}$

Approach
MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

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

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**: $\{c_1\}$, $\{c_4\}$, $\{c_5\}$,$\{c_2, c_3\}$

Approach

MCS: Minimal Correction Subset

MCS: Definition

Let C an **infeasible** set of constraints

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

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**: $\{c_1\}$, $\{c_4\}$, $\{c_5\}$, $\{c_2, c_3\}$

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**: $\{c_1\}$, $\{c_4\}$, $\{c_5\}$,$\{c_2, c_3\}$

Approche
(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 **chemin** 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**

Approche

(`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 **chemin** 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**

Example

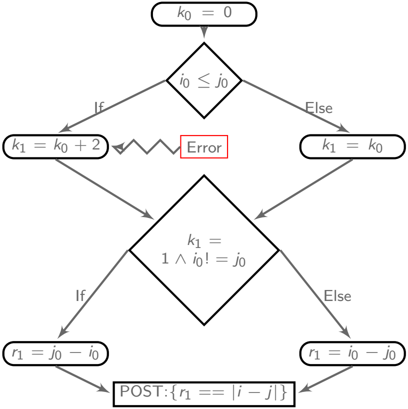
Calculate the absolute value of i-j

```
class AbsMinus {
/*returns|i-j|,the absolute value of i minus j*/
/*@ ensures
  @ (result==|i-j|);
  @*/
  int 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;
    }
    return result;
  }
}
```

Example

Calculate the absolute value of i-j

```
class AbsMinus {
/*returns|i-j|,the absolute value of i minus j*/
/*@ ensures
  @ (result==|i-j|);
  @*/
  int 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;
    }
    return result;
  }
}
```



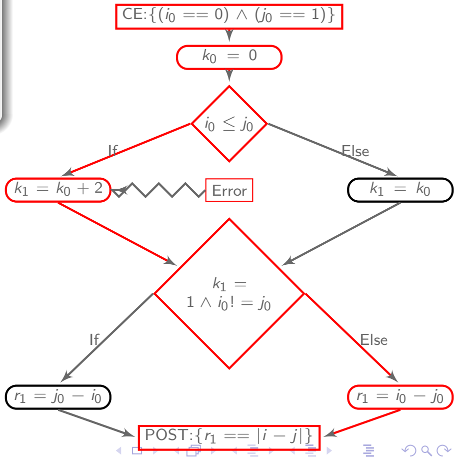
Example

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

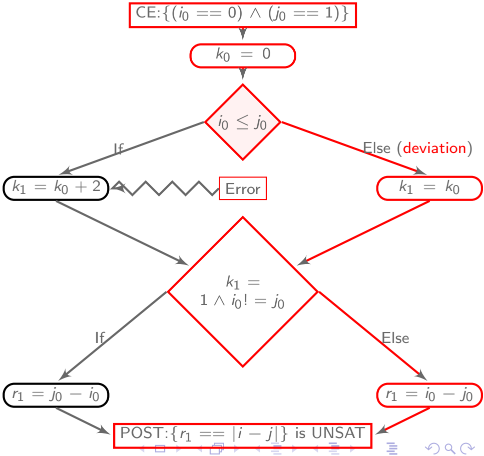
The **path** obtained **by deviating** the condition $i_0 \leq j_0$

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 : $\{(k_1 = 1 \wedge i_0! = j_0)\}$

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

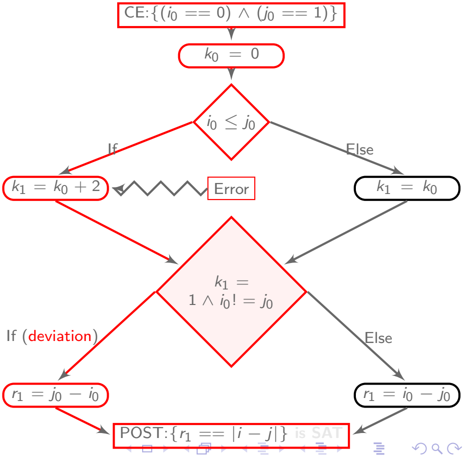
$P \cup \{r_1 = |i - j|\}$ **is consistent**

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

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

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

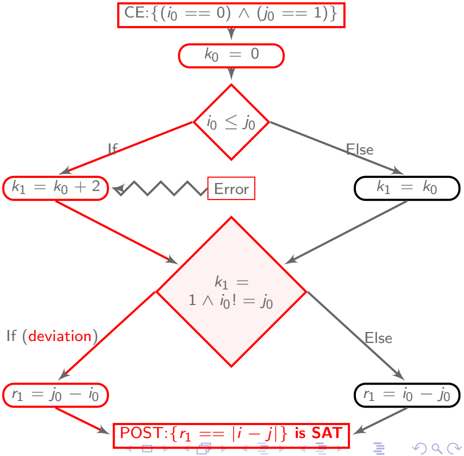
$P \cup \{r_1 = |i - j|\}$ **is consistent**

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 : $\{k_0 = 0\}, \{k_1 = k_0 + 2\}$



State of the art

Approaches based on systematic testing

Tarantula, Delta Debugging

- 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

State of the art

Approaches based on systematic testing

Tarantula, Delta Debugging

- 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

State of the art

Approaches based on systematic testing

Tarantula, Delta Debugging

- 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

State of the art

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

State of the art

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

Experimental evaluation

Tools used

- **LocFaults**: our implementation
  → The MIP solver of IBM ILOG 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

- **Simple programs**
  - → AbsMinus, Minmax, Mid (illustrative programs)
  - → Maxmin6var (program without junctions)
  - → Tritype, TriPerimetre (programs with junctions)
  - → Several erroneous versions for each program

    Example: TriPerimetre → TriPerimetreKO, TriPerimetreKO2, TriPerimetreKO3
- **TCAS(Traffic Collision Avoidance System), a realistic benchmark**
  - → 1608 test cases, except cases for overflow *PositiveRAAltThresh* table
  - → TcasKO . . . TcasKO41

Experimental evaluation

Programs built

- **Simple programs**
  - → AbsMinus, Minmax, Mid (illustrative programs)
  - → Maxmin6var (program without junctions)
  - → Tritype, TriPerimetre (programs with junctions)
  - → Several erroneous versions for each program

    Example: TriPerimetre → TriPerimetreKO, TriPerimetreKO2, TriPerimetreKO3
- **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	Counterexample	LocFaults = 0	LocFaults ≤ 1	LocFaults ≤ 2	LocFaults ≤ 3	BugAssist
AbsMinusKO3	$\{i = 0, j = 1\}$	{20}	{16},{14},{12} {20}	{16},{14},{12} {20}	{16},{14},{12} {20}	{16, 20}
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	{10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18, 19, 22}
MidKO	$\{a = 2, b = 1, c = 3\}$	{19}	{19}	{19}	{14, 23, 26} {19}	{14, 19, 30}
Maxmin6varKO4	$\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$	{116}	{116}	{116}	{12, 15, 19} {116}	{12, 166}
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	{54}	{21} {26} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27},{25} {32, 44},{33},{25},{27} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27},{25} {32, 44},{33},{25},{27} {35},{27},{25} {53},{25},{27} {54}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 53, 68}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	{46}	{45},{33},{25} {46}	{26, 32} {29, 32} {45},{33},{25} {46}	{26, 32} {29, 32} {32, 35, 49},{25} {32, 35, 53},{25} {32, 35, 57},{25} {45},{33},{25} {46}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	{58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{28, 29, 31, 32, 35, 37, 65, 72}

LocFaults provides a more informative and explanatory localization

Experimental evaluation

Results (time calculation)

Program	LocFaults					BugAssist	
	P	L				P	L
		$=0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO3	$0,479s$	$0,076s$	$0,113s$	$0,357s$	$0,336s$	$0,02s$	$0,04s$
MinmaxKO	$0,528s$	$0,243s$	$0,318s$	$0,965s$	$1,016s$	$0,01s$	$0,09s$
MidKO	$0,524s$	$0,065s$	$0,078s$	$0,052s$	$0,329s$	$0,02s$	$0,08s$
Maxmin6varKO4	$0,538s$	$0,06s$	$0,07s$	$0,075s$	$0,56s$	$0,04s$	$0,78s$
TritypeKO2	$0,51s$	$0,023s$	$0,25s$	$2,083s$	$3,864s$	$0,02s$	$0,69s$
TritypeKO4	$0,497s$	$0,023s$	$0,095s$	$0,295s$	$5,127s$	$0,02s$	$0,21s$
TriPerimetreKO	$0,518s$	$0,047s$	$0,126s$	$1,096s$	$2,389s$	$0,03s$	$0,64s$

The times of `LocFaults` are close to the times of `BugAssist`

Experimental evaluation

Results (Number of errors localized for TCAS)

Programme	Nb_E	Nb_CE	LF ≤ 1	LF ≤ 2	BA
TcasKO	1	131	131	131	131
TcasKO2	2	67	67	134	67
TcasKO3	1	23	2	2	23
TcasKO4	1	20	16	20	20
TcasKO5	1	10	10	10	10
TcasKO6	3	12	36	36	24
TcasKO7	1	36	23	36	0
TcasKO8	1	1	1	1	0
TcasKO9	1	7	7	7	7
TcasKO10	6	14	16	65	84
TcasKO11	6	14	16	34	46
TcasKO12	1	70	52	52	70
TcasKO13	1	4	3	3	4
TcasKO14	1	50	6	6	51
TcasKO16	1	70	22	70	0
TcasKO17	1	35	22	35	0
TcasKO18	1	29	21	28	0
TcasKO19	1	19	13	19	0
TcasKO20	1	18	18	18	18
TcasKO21	1	16	16	16	16
TcasKO22	1	11	11	11	11
TcasKO23	1	41	41	41	41
TcasKO24	1	7	7	7	7
TcasKO25	1	3	0	3	3
TcasKO26	1	11	11	11	11
TcasKO27	1	10	10	10	10
TcasKO28	2	75	74	148	121
TcasKO29	2	18	17	35	0
TcasKO30	2	57	57	114	0
TcasKO34	1	77	77	77	77
TcasKO35	4	75	74	148	115
TcasKO36	1	122	120	120	0
TcasKO37	4	94	110	235	236
TcasKO39	1	3	0	3	3
TcasKO40	2	122	0	103	120
TcasKO41	1	20	17	20	20

The performances of `LocFaults` are favorably comparable to `BugAssist`

Conclusion

- Bounded approach for error localization
  - **Bounded DFS** exploration
  - **Bounded MCS** calculation

    → To prevent the combinatorial explosion
- Based on the use of constraint solvers
- `LocFaults` provides more accurate and relevant results compared to `BugAssist`

  → The times of the two tools are similar
- `LocFaults` locates errors frequently for the TCAS programs

Conclusion

- Bounded approach for error localization
  - **Bounded DFS** exploration
  - **Bounded MCS** calculation

    → To prevent the combinatorial explosion
- Based on the use of constraint solvers
- LocFaults provides more accurate and relevant results compared to BugAssist

  → The times of the two tools are similar
- LocFaults locates errors frequently for the TCAS programs

Conclusion

- Bounded approach for error localization
  - **Bounded DFS** exploration
  - **Bounded MCS** calculation

    → To prevent the combinatorial explosion
- Based on the use of constraint solvers
- `LocFaults` provides more accurate and relevant results compared to `BugAssist`

  → The times of the two tools are similar
- `LocFaults` locates errors frequently for the TCAS programs

Conclusion

- Bounded approach for error localization
  - **Bounded DFS** exploration
  - **Bounded MCS** calculation

    → To prevent the combinatorial explosion
- Based on the use of constraint solvers
- `LocFaults` provides more accurate and relevant results compared to `BugAssist`

  → The times of the two tools are similar
- `LocFaults` locates errors frequently for the TCAS programs

Outlook

- Incremental version of the algorithm
- Treat programs with loops
- Extend our implementation to support more complex non-linear instructions through the use of **specialized solvers**
- Experiment more of real programs
- Treat the programs with floating-point numbers computation

  $\rightarrow$ Use **specialized solvers** on floating computations

Outlook

- Incremental version of the algorithm
- Treat programs with loops
- Extend our implementation to support more complex non-linear instructions through the use of **specialized solvers**
- Experiment more of real programs
- Treat the programs with floating-point numbers computation

  $\rightarrow$ Use **specialized solvers** on floating computations

Outlook

- Incremental version of the algorithm
- Treat programs with loops
- Extend our implementation to support more complex non-linear instructions through the use of **specialized solvers**
- Experiment more of real programs
- Treat the programs with floating-point numbers computation

  $\rightarrow$ Use **specialized solvers** on floating computations

Outlook

- Incremental version of the algorithm
- Treat programs with loops
- Extend our implementation to support more complex non-linear instructions through the use of **specialized solvers**
- Experiment more of real programs
- Treat the programs with floating-point numbers computation

  → Use **specialized solvers** on floating computations

Outlook

- Incremental version of the algorithm
- Treat programs with loops
- Extend our implementation to support more complex non-linear instructions through the use of **specialized solvers**
- Experiment more of real programs
- Treat the programs with floating-point numbers computation

  → Use **specialized solvers** on floating computations

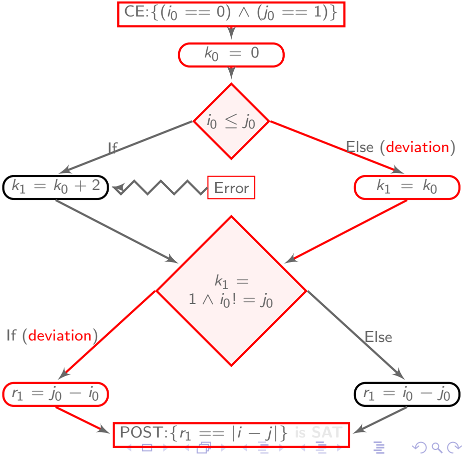
Thank you for your attention.

Questions ?

Example

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

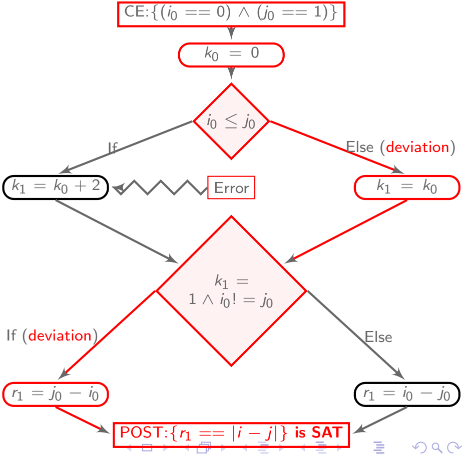
The **deviated** conditions :
$\{i_0 \leq j_0, (k_1 = 1 \wedge i_0 != j_0)\}$
$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\}$
$P \cup \{r_1 = |i - j|\}$ **is consistent**
The deviation is not minimal



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



Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
$C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
while $\underline{\mathrm{SAT}(C') \land k \leq MCS_b}$ **do**
  $C'_k \leftarrow C' \land \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$
  while $\underline{\mathrm{SAT}(C'_k)}$ **do**
    $newMCS \leftarrow \emptyset$
    forall the $\underline{\text{indicator } y_i}$ **do**
      Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.
      si $\underline{val(y_i) = 0}$ **alors**
        $newMCS \leftarrow newMCS \cup \{c_i\}$.
      fin
    end
    $MCS.add(newMCS)$.
    $C'_k \leftarrow C'_k \land \textsc{BlockingClause}(newMCS)$
    $C' \leftarrow C' \land \textsc{BlockingClause}(newMCS)$
  end
  $k \leftarrow k + 1$
end

return MCS

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
$C' \leftarrow$ AddYVars(C); $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
while $\underline{\mathrm{SAT}(C') \wedge k \leq MCS_b}$ **do**
  $C'_k \leftarrow C' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$,$k$)
  while $\underline{\mathrm{SAT}(C'_k)}$ **do**
    $newMCS \leftarrow \emptyset$
    forall the $\underline{\text{indicator } y_i}$ **do**
      Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.
      si $\underline{val(y_i) = 0}$ **alors**
        $newMCS \leftarrow newMCS \cup \{c_i\}$.
      fin
    end
    $MCS.add(newMCS)$.
    $C'_k \leftarrow C'_k \wedge$ BlockingClause($newMCS$)
    $C' \leftarrow C' \wedge$ BlockingClause($newMCS$)
  end
  $k \leftarrow k + 1$
end

return MCS

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
$C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
while $\underline{\mathrm{SAT}(C') \wedge k \leq MCS_b}$ **do**
    $C'_k \leftarrow C' \wedge \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$
    while $\underline{\mathrm{SAT}(C'_k)}$ **do**
        $newMCS \leftarrow \emptyset$
        forall the $\underline{\text{indicator } y_i}$ **do**
            Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.
            si $\underline{val(y_i) = 0}$ **alors**
                $newMCS \leftarrow newMCS \cup \{c_i\}$.
            fin
        end
        $MCS.add(newMCS)$.
        $C'_k \leftarrow C'_k \wedge \textsc{BlockingClause}(newMCS)$
        $C' \leftarrow C' \wedge \textsc{BlockingClause}(newMCS)$
    end
    $k \leftarrow k + 1$
end

return MCS

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)

Inputs C: Infeasible set of constraints, MCS_b: integer

Outputs MCS: List of MCS in C of a cardinality less than MCS_b

$C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;

while $\underline{\mathrm{SAT}(C') \wedge k \leq MCS_b}$ **do**

  $C'_k \leftarrow C' \wedge \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$

  while $\underline{\mathrm{SAT}(C'_k)}$ **do**

    $newMCS \leftarrow \emptyset$

    forall the indicator y_i **do**

      Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.

      si $\underline{val(y_i) = 0}$ **alors**

        $newMCS \leftarrow newMCS \cup \{c_i\}$.

      fin

    end

    $MCS.add(newMCS)$.

    $C'_k \leftarrow C'_k \wedge \textsc{BlockingClause}(newMCS)$

    $C' \leftarrow C' \wedge \textsc{BlockingClause}(newMCS)$

  end

  $k \leftarrow k + 1$

end

return MCS

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

```
Function MCS(C, MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
C' ← AddYVars(C); MCS ← ∅; k ← 1;
while SAT(C') ∧ k ≤ MCS_b do
    C'_k ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n}, k)
    while SAT(C'_k) do
        newMCS ← ∅
        forall the indicator y_i do
            Let y_i indicator of the constraint c_i ∈ C, and val(y_i) the value of y_i in the solution
            calculated of C'_k.
            si val(y_i) = 0 alors
                newMCS ← newMCS ∪ {c_i}.
            fin
        end
        MCS.add(newMCS).
        C'_k ← C'_k ∧ BlockingClause(newMCS)
        C' ← C' ∧ BlockingClause(newMCS)
    end
    k ← k + 1
end

return MCS
```

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
$C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
while $\text{SAT}(C') \land k \leq MCS_b$ **do**
 $C'_k \leftarrow C' \land \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$
 while $\text{SAT}(C'_k)$ **do**
  $newMCS \leftarrow \emptyset$
  forall the indicator y_i **do**
   Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.
   si $val(y_i) = 0$ **alors**
    $newMCS \leftarrow newMCS \cup \{c_i\}$.
   fin
  end
  $MCS.add(newMCS)$.
  $C'_k \leftarrow C'_k \land \textsc{BlockingClause}(newMCS)$
  $C' \leftarrow C' \land \textsc{BlockingClause}(newMCS)$
 end
 $k \leftarrow k + 1$
end

return MCS

Approach

The algorithm of Liffiton and Sakallah to compute the MCS subsets

Function MCS(C,MCS_b)
Inputs C: Infeasible set of constraints, MCS_b: integer
Outputs MCS: List of MCS in C of a cardinality less than MCS_b
$C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
while $\mathrm{SAT}(C') \wedge k \leq MCS_b$ **do**
    $C'_k \leftarrow C' \wedge \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$
    while $\mathrm{SAT}(C'_k)$ **do**
        $newMCS \leftarrow \emptyset$
        forall the indicator y_i **do**
            Let y_i indicator of the constraint $c_i \in C$, and $val(y_i)$ the value of y_i in the solution calculated of C'_k.
            si $val(y_i) = 0$ **alors**
                $newMCS \leftarrow newMCS \cup \{c_i\}$.
            fin
        end
        $MCS.add(newMCS)$.
        $C'_k \leftarrow C'_k \wedge \textsc{BlockingClause}(newMCS)$
        $C' \leftarrow C' \wedge \textsc{BlockingClause}(newMCS)$
    end
    $k \leftarrow k + 1$
end

return MCS