On the Capabilities of CP for Numerical Program Analysis

Michel Rueher

Joined work with

Hélène Collavizza, Claude Michel, Olivier Ponsini, Pascal Van Hentenryck, Mohammed Said Belaid, Le Vinh Nguyen, Mohammed Bekkouche

University of Nice Sophia Antipolis – CNRS

Master Class “CONSTRAINT PROGRAMMING AND VERIFICATION »

CPAIOR 2015, Barcelona, May 2015

Outline

- **BMC** (Bounded Model Checking)
  - Goal : *Finding counter-examples violating an assertion*
  - State of the art Methods → *SAT /SMT Solvers*

- **Program analysis**
  - Goal: *Get rid of false alarms*
  - State of the art Methods → *abstract interpretation*

- **Fault localization**
  - Goal: *locations of potentially faulty statements*
  - State of the art Methods → *MaxSat*

Bounded Model Checking

- **Context**: programs with ***numeric operations*** over integer or floating point numbers

- **Goal** : *Finding counter-examples violating an assertion*

Bounded Model Checking framework

Models → finite automates, labelled transition systems

Properties:

- Safety → something bad should not happen
- Liveness → something good should happen

Bound *k* → look only for counter examples made of *k* states

Bounded Model Checking framework (cont.)

% set of states: S, initial states: I, transition relation: T

% bad states B reachable from I via T?

bounded_model_checker$_{forward}$(I,T,B,k)

SC =∅; SN =I; n=1

while $S_C \neq S_N$ **and** n<k **do**

If $B \cap S_N \neq \emptyset$

then return *"found error trace to bad states"*;

else $S_C = S_N$; $S_N = S_C \cup T(SC)$; $n = n + 1$;

done

return *"no bad state reachable"*;

SAT/SMT - Based BMC framework

1 The *program is unwound k* times

2 The unwound (and simplified) *program* and the *negation* of the property are translated into *a big propositional formula* φ

φ is satisfiable iff there exists a counterexample of depth less than k

SAT solvers solvers have a *"**Global** view"*

Numerical expressions → **Boolean abstraction**

→ **Spurious solutions**

Critical issue: ***relevant minimum conflict sets*** to ***limit backtracks***

CP-Based BMC framework

1 The ***program is unwound k*** times

2 The unwound (and simplified) program in ***SSA/DSA form*** and the ***negation property*** are translated ***on the fly*** into ***constraint system Cs***

Cs is satisfiable for a full path iff there exists a counterexample of depth less than k

Various solvers and ***strategies*** can be used

To explore only a limited part of the search space, ***efficient pruning is a critical issue***

CP-Based BMC: CPBPV, a depth first strategy

CPBPV:

- Translate ***precondition*** (if exists) and ***property*** to check into a set of constraints
- ***Explore each branch Bi*** of the program and translate statements of branch *Bi* into a set of constraints
  - If for each branch *Bi,* the generated ***CSP is inconsistent*** , then the ***program is conform*** with its specification
  - If for some branch *Bi* the generated ***CSP has a solution,*** then this solution is a ***counterexample*** → exhibits ***a non-conformity***

Inconsistencies are detected at each node of the control flow graph

CP-Based BMC: DPVS, a Dynamic Backjumping Strategy

Start from the post-condition and jump to the first locations where the variables of the post-condition are assigned

Essential observation:

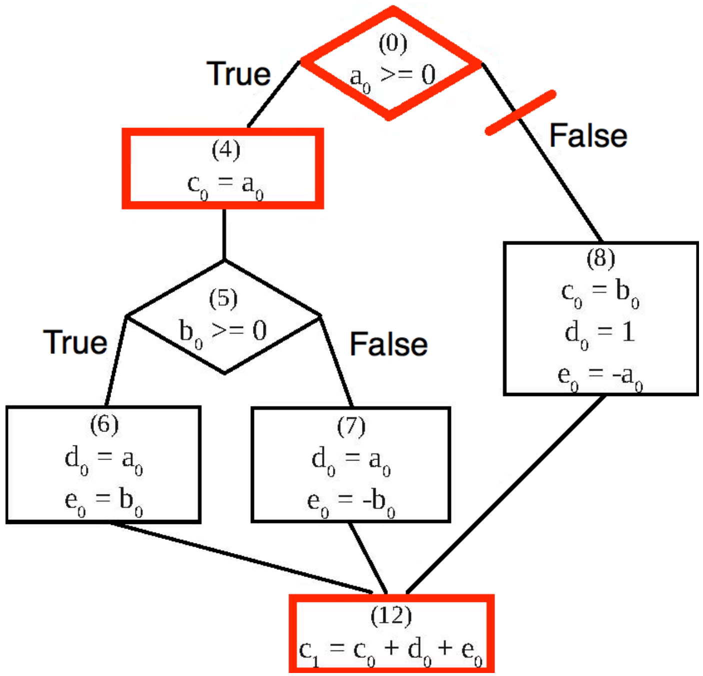
When the program is in an SSA-like form, CFG does not have to be explored in a top down (or bottom up) way

→ ***compatible blocks can just be collected in a non-deterministic way***

Why does it pay off ?

- **Enforces the constraints** on the domains of the selected variables
- **Detects inconsistencies earlier**

CP-Based BMC: DPVS, example



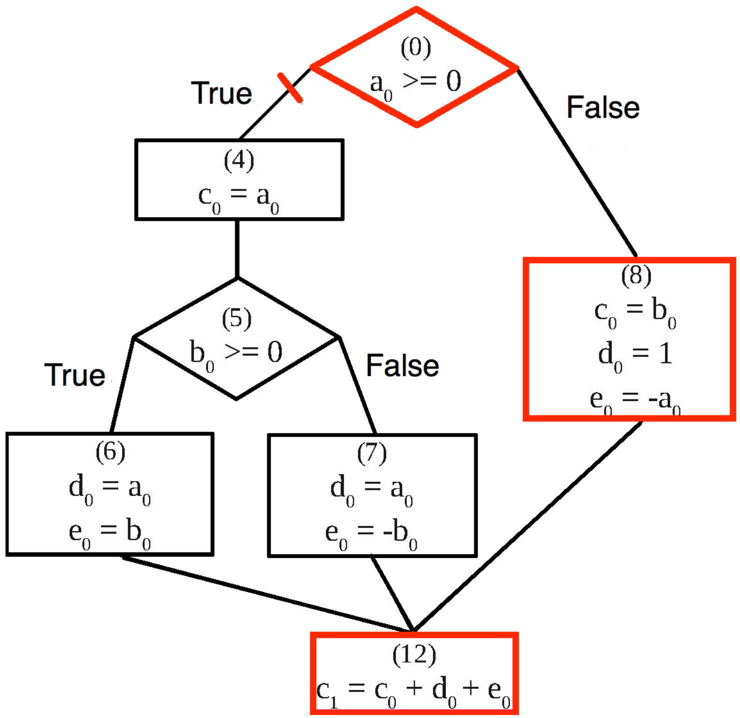

```
void foo(int a, int b)
int c, d, e, f;
if(a>=0) {
        if (a<10)  {f=b-1;}
        else {f=b-a;}
        c=a;
        if (b>=0) {d=a; e=b}
        else {d=a; e=-b;}    }
else {
        c=b; d=1; e=-a;
        if (a>b) {f=b+e+a;}
        else {f=e*a-b;}  }
c = c+ d + e;
assert(c>=d+e); // property p1
assert(f>=-b*e); // property p2
```

To prove property p_1, select node (12), then select node (4)

→ the condition in node (0) must be true

$S=\{c_1 < d_0+e_0 \wedge c_1 = c_0+d_0+e_0 \wedge c_0 = a_0 \wedge a_0 \geq_0\} = \{a_0 < 0 \wedge a_0 \geq 0\}$ ... **inconsistent**

CP-Based BMC: DPVS, example (cont.)



void foo(**int** a, **int** b)
int c, d, e, f;
if(a>=0) {
    ~~**if** (a<10) {f=b-1;}~~
    ~~**else** {f=b-a;}~~
    c=a;
    if (b>=0) {d=a; e=b}
    else {d=a; e=-b;} }
else {
    c=b; d=1; e=-a;
    ~~**if** (a>b) {f=b+e+a;}~~
    ~~**else** {f=e*a-b;} }~~
c = c+ d + e;
assert(c>=d+e); // property p_1
assert(f>=-b*e); // property p_2

Select node (8) → condition in node (0) must be false:

$S = \{c1 < d_0 + e_0 \wedge c1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0\}$

$\quad = \{a_0 < 0 \wedge b_0 < 0\}$

Solution $\{a0 = -1, b0 = -1\}$

CP-Based BMC: Static versus Dynamic Strategies

Two benchmarks:

- **Flasher Manager**, industrial application
- **Binary Search**

Bench	DPVS	CPBPV
FM 5	0.5	1.24
FM 100	15.95	> **600**
FM 200	**22.65**	> 600
BS 8	35	0.2
BS 16	> **600**	**1.14**

→ ***Pruning is a critical issue***

CP-Based program analysis

- **Context**:
  - **Embedded Systems** (Anti-lock Braking System controller, ...) rely more and more on floating-point computations
  - C language is widely used for such applications (often C code generated from a Simulink model)

    Floats $\longrightarrow$ a source of **execution errors**

- **Goal:** ***Get rid of false alarms*** *(generated by abstract interpretation tools)*

Problems with floating-point numbers

Rounding: ***Counter-intuitive properties***

- Arithmetic operators are neither associative nor distributive
- Reasoning with ***absorption*** and ***cancellation***

Examples (in simple precision, binary representation):

- Absorption: $10^7 + 0.5 = 10^7$
- Cancellation: $((1 - 10^{-7}) - 1) * 10^7 = -1.192...(\neq -1)$
- $(10000001-10^7)+0.5 \neq 10000001-(10^7+0.5)$
- $0.1=(0.000110011001100...)$

Problems with floating-point numbers (cont.)

Programs are run on the floats **but**:

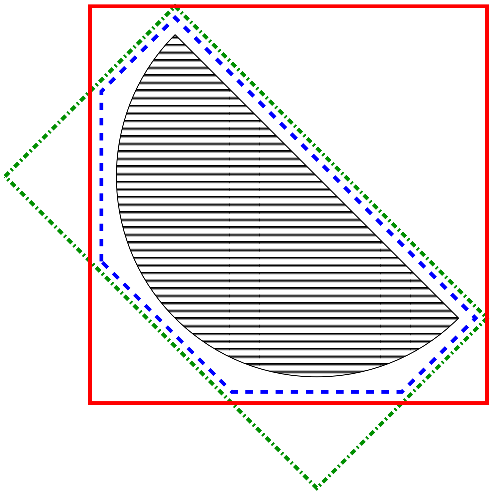
- **Specification, properties** of programs

  $\longrightarrow$ Users are **reasoning with real numbers**

- **Programs** are often written with the semantics of real numbers "in mind"
- **Differences** between computations over real numbers and computations over the floats

  $\longrightarrow$ ***Execution problems on programs with floats***

Abstract Interpretation



Goal: static detection of execution errors

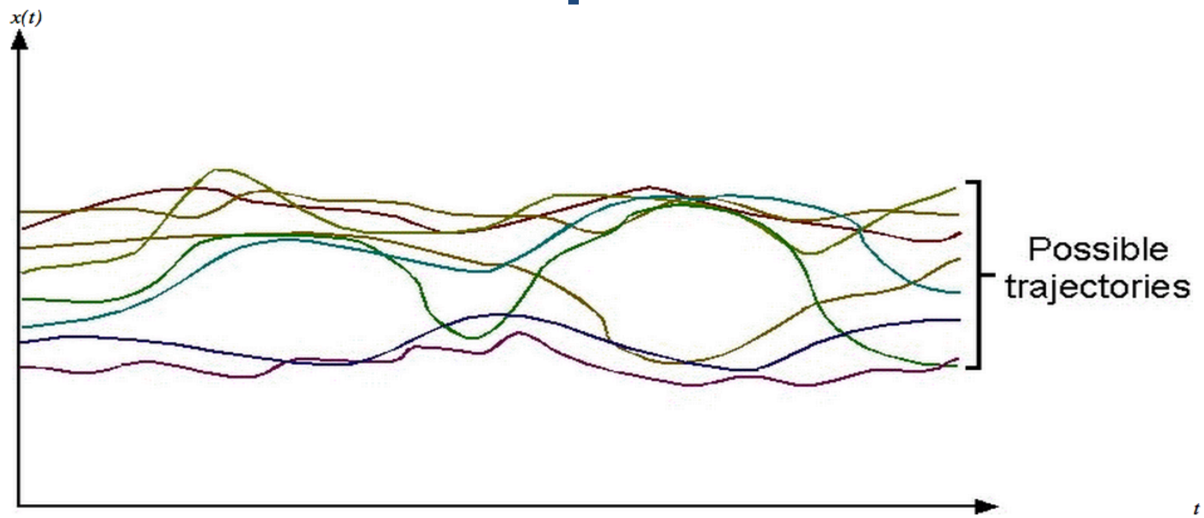
→ Approximations of computations over floats and of computations over the real numbers

Intervals, **zonotopes**, **polyhedra**...

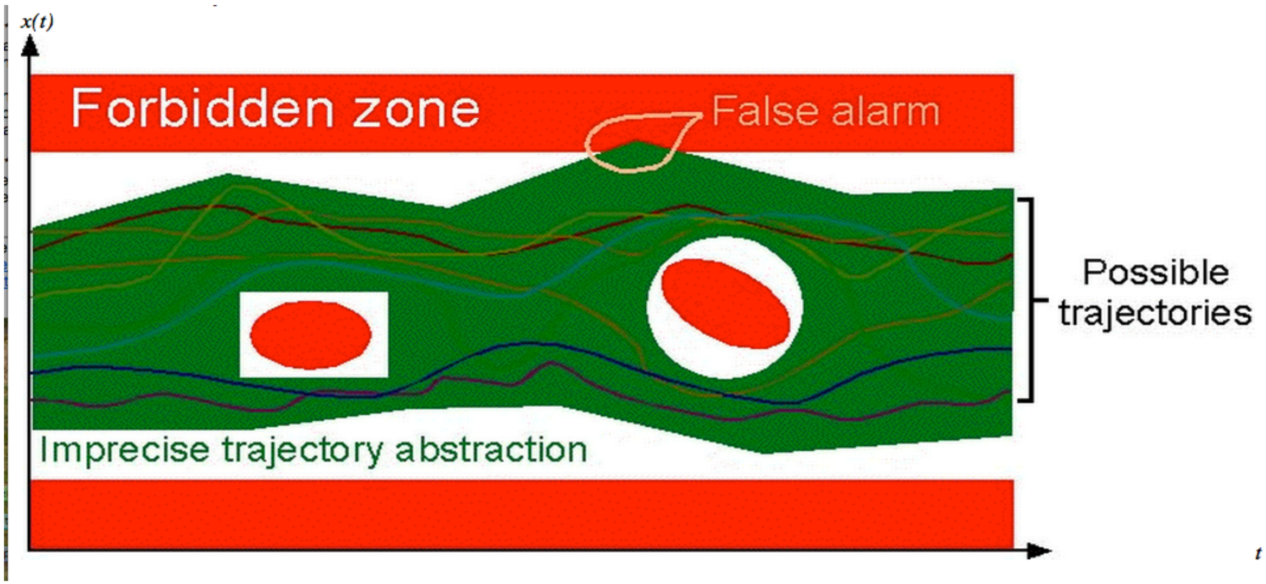
Zonotopes: convex polytopes with a central symmetry (sets of affine forms)

- \+ **Good trade-off** between performance and precision
- – **Not very accurate** for nonlinear expressions and on very common program constructs such as conditionals

Limits of Abstract Interpretation: false alarms



Courtesy
to Patick Cousot



AI versus CP

Abstract Interpretation: ***good scalability*** for estimating rounding errors but *over-approximation*

→ ***false alarms***

→ *totally* ***inappropriate behaviours*** *of a program may be dreaded but the developer does not know whether these behaviours will actually occur !*

Constraint Programming :

Good precision (strong refutation capabilities, finding counter examples) but ***lack of scalability***

rAICP: Combining AI and CP (cont.)

Successive exploration and merging steps

- Use of ***AI to compute a first approximation*** of the values of variables at a program node where two branches join

- Building a constraint system for each branch between two join nodes in the CFG of the program and use of ***CP local consistencies to shrink the domains*** computed by AI

rAICP: example

```
1 /* Pre-condition:   f,g ∈ [-10,10]  */
2 float foo(float f, float g) {
3   float x, y, z;
4 
5   x = f + 2 * g;
6 
7   if (x <= 0) {
8     y = g;
9   }
10  else {
11    y = -g;
12  }
13 
14  if (y >= 0) {
15    z = 10*y;
16  }
17  else {
18    z = -y;
19  }
20 
21  return z;
22 }
```

On floats and reals, foo → z= [0,50]

Fluctuat → z= [0,100]

Merge points of foo: lines 13 and 21

lines 1 →13:

Fluctuat → $f,g,y \in [-10,10]$, $x \in [-10, 0]$

FPCS (path 1, "then" branch) :

$C = \{x = f + 2*g \wedge x \leq 0 \wedge y = g \wedge -10 \leq f \wedge f \leq 10 \wedge -10 \leq g \wedge g \leq 10 \wedge -10 \leq y \wedge y \leq 10 \wedge -10 \leq x \wedge x \leq 0\}$

$\rightarrow g,y \in [-10,5]$

lines 14 → 22:

Fluctuat → $z \in [0,50]$

rAICP: Filtering techniques

FPCS: ***solver over floating-point constraints*** combining ***interval propagation*** with explicit ***search***

- **Correct** solver over the floats : ***no solutions are lost***
- Based on ***2B-consistency*** and ***3B-consistency***

  Projection functions for floats:

  - *Direct projections*: straightforward adaptation of interval arithmetic
  - Inverse projections: less intuitive, more complex (e.g., might need a larger format than the system variables)
- ***Handling of rounding modes***, nonlinear expressions and the usual mathematical functions (trigonometric. . .)

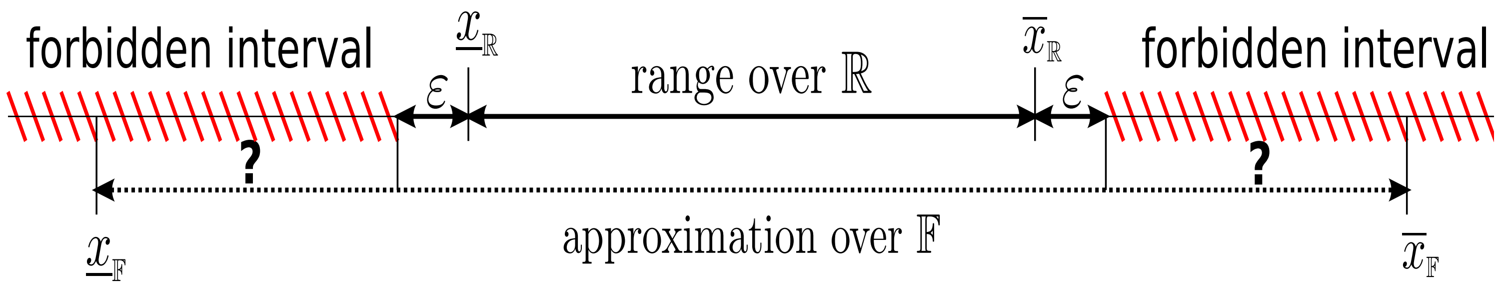
Experiments: eliminating false alarms

CDFL: Program analyser for proving the absence of runtime errors in program with floating-point computations based on *Conflict-Driven Learning*

	rAICP	**Fluctuat**	**CDFL**
False alarms	**0**	**11**	**0**
Total time	40.55s	**18.33 s**	**208.99** s

Computed on the 55 benchs from CDFL paper (TACAS'12, *D'Silva, Leopold Haller, Daniel Kroening, Michael Tautschnig*)

Generating Test Cases inside Suspicious Intervals



- **Suspicious intervals** for *x* : $[\underline{\mathbf{x}}_F , \underline{\mathbf{x}}_R - \varepsilon]$ or $[\mathbf{x}_R + \varepsilon, \mathbf{x}_F]$
- Tolerance specified by the user : ε
- **Question:** *Can the program hit a forbidden zone over the floating-point numbers?*

Proposed approach : CPBPV_FP

"Forward" propagation

Computing the suspicious interval of *x*

→ approximate the domain of x over the reals by

→ approximate domain of x over the floats by $[\underline{\mathbf{x}}_F, \mathbf{x}_F]$

"Backward " propagation

Computing test-cases inside a suspicious interval of *x*

→ Solving a bounded-model checking problem with domain of x restricted to $[\underline{\mathbf{x}}_F, \underline{\mathbf{x}}_R - \varepsilon]$ or $[\mathbf{x}_R + \varepsilon, \mathbf{x}_F]$

CPBPV_FP : CP based BMC for floats

Outputs :

- **A test case**

  $\longrightarrow$ *P* ***can produce*** a suspicious value for *x*
- **A proof that no test case exists**

  $\longrightarrow$ the suspicious interval ***can be removed***

  Only the case if the loops in P cannot be unfolded beyond the bound k
- **An inconclusive answer**

  $\longrightarrow$ *P* ***may produce*** a suspicious value

no test case could be generated
but the loops in P could be unfolded beyond the bound k

FPCS Search Strategies

- **std:** standard *prune & bisection-based search*
- **fpc:** domain of selected variable is split in *5 intervals*
  - 3 degenerated intervals: the smallest float ***l***, the largest float ***r***, and the mid-point ***m***
  - intervals ***]l, m[*** and ***]m, r[***
- **fp3s:** domain of selected variable is split in 3 degenerated intervals: the smallest float ***l***, the largest float ***r***, and the mid-point ***m***

Experiments: tools

- **CDFL:** Program analyser for proving the absence of runtime errors in program with floating-point computations based on Conflict-Driven Learning

- **CBMC:** state of art bounded mode checkers

- **CPBPV_FP:** our constraint-based bounded- model checking framework

Experiments: Program Heron

Uses Heron's formula to compute the area of a triangle from the lengths of a, b, and c (a being the longest side):

area= sqrt(s*(s-a)*(s-b)*(s-c) with **s=(a+b+c)/2**

```
/* Pre-condition :  a ≥ b and a ≥ c */
float heron(float a, float b, float c) {
  float s, squared_area;

  squared_area = 0.0f;
  if (a <= b + c) {
    s = (a + b + c) / 2.0f;
    squared_area = s*(s-a)*(s-b)*(s-c);
  }
  return sqrt(squared_area);
}
```

Optimized Heron : squared_area = ((a+(b+c))*(c−(a−b))*(c+(a−b))*(a+(b−c)))/16.0f;

Experiments

Name	Condition	CDFL	CBMC	std	fpc	fpc3s	s?
heron	aera < 10_f^{-5} area > 156.25f +10_f^{-5}	3.87 s > 180 s	0.28 s 34.51 s	> 180 s 22. 32 s	0.70 s 7.80 s	0.02 s n 0.08 s n	y y
optimized heron	aera < 10_f^{-5} area > 156.25f +10_f^{-5}	7.61 s > 180 s	0.93 s > 180 s	> 180 s 8.99 s	0.15 s 30.48 s	0.01 s n 0.01 s n	y n

std: standard *prune & bisection-based search*

fpc: domain of selected variable is split in *5 intervals*

- *3 degenerated intervals*: the smallest float ***l***, the largest float ***r***, and the mid-point ***m***
- intervals ***]l, m[*** and ***]m, r[***

fp3s: domain of selected variable is split in 3 degenerated intervals: the smallest float ***l***, the largest float ***r***, and the mid-point ***m***

Fault localization

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

- **Goal:** Provide helpful information for *error localization on numeric constraint systems*
- **Input:**
  - Some imperative program with *numeric statements* (over integers or floating-point numbers)
  - *An assertion* to be checked
  - A *counter-example* that violates the assertion
- **Output**: information on locations of *potentially faulty statements*

Fault localization – Keys 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* → ***M****aximal* ***S****atisfiable* ***S****ubset*
  - *Computing Minimal sets of statements to withdraw* → ***M****inimal* ***C****orrection* ***S****et ?*

MSS, MCS: Definitions

- **MSS** Maximal Satisfiable Subset

  a generalization of MaxSAT considering maximality instead of maximum cardinality

  $M \subseteq C$ is a MSS $\Leftrightarrow$ M is SAT and $\forall c_i \in C \setminus M : M \cup \{c_i\}$ 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 $\forall c_i \in M : (C \setminus M) \cup \{c_i\}$ is UNSAT

LocFaults – Selecting Diverted Paths

- **Explore the path of the counter-example** and ***paths with at most** k* ***deviations***

  Example: one deviation

  Decision for one conditional statement is switched and the input data of the counter-example are propagated → new path **P'**

  Iff $CSP_{P'} \cup CSP_{POST}$ ***is satisfiable, MCS are computed for P'***

- **Compute MCS** with ***at most m suspicious statements***

  Bounds k and m are mandatory because there are an exponential number

  of paths and sets of suspicious statements

LocFaults – Computing MCCs for Diverted Paths

Let be:

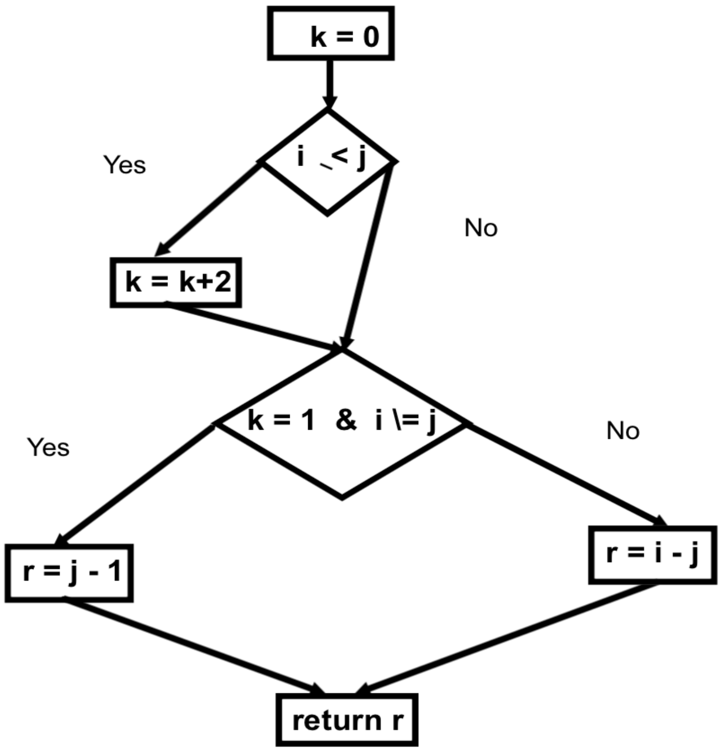
- **P**, a path generated by **k decision switches** of conditional statements **$cond_1$, .., $cond_k$** and by the propagation of **CE**
- **C**, the constraints of **P**, and **C_k**, the constraints generated by the assignments **occurring before $cond_k$** along **P_k**

If **C ∪ POST holds**:

- **{¬$cond_1$, .., ¬$cond_k$ }** is **a potential correction,**
- The **MCS** of **C_k ∪ {¬$cond_1$, .., ¬$cond_k$ }** are **potential corrections**

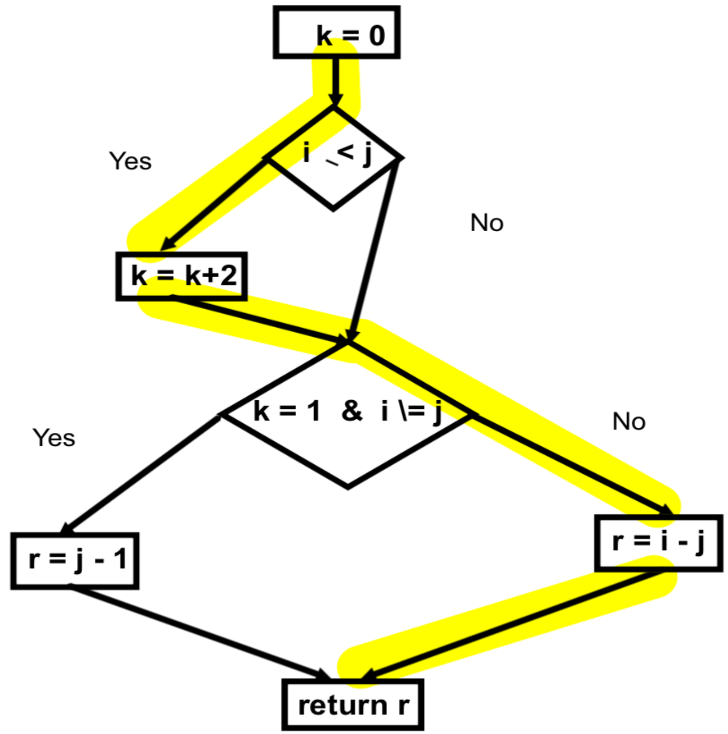
Note: {¬*cond*$_1$, .., ¬*cond*$_k$ } is a ”hard” constraint

LocFaults – Exemple



CFG of AbsMinus

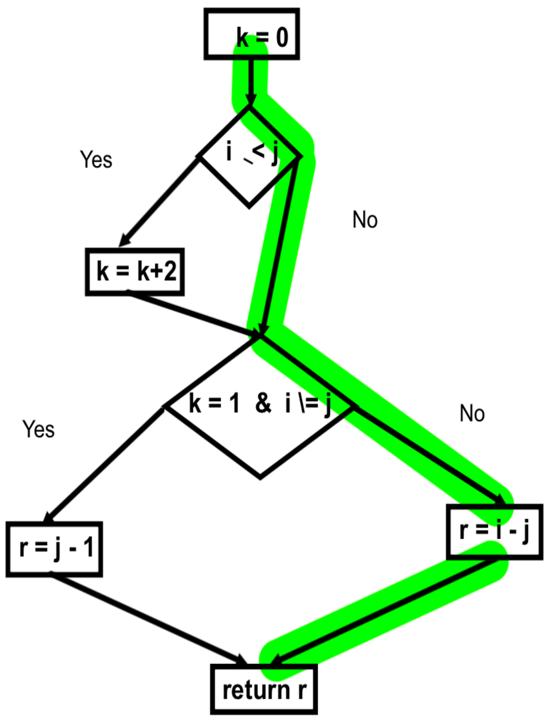
CE: {i =0,j =1}

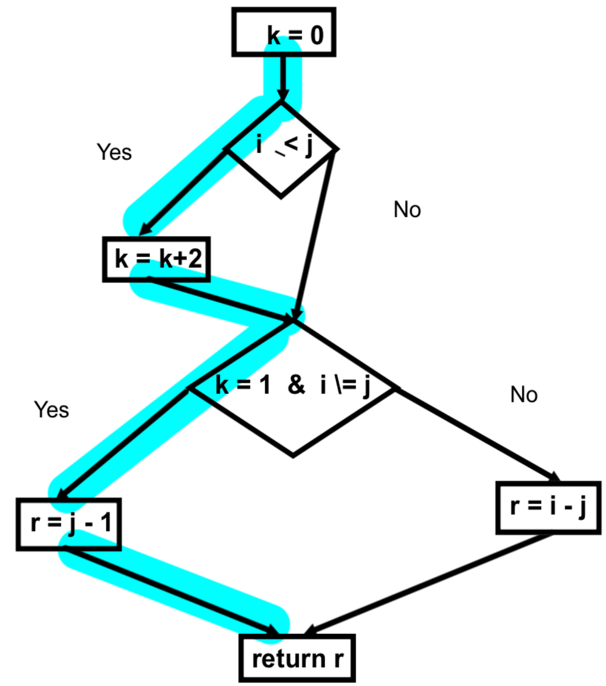


Faulty path for {i = 0, j = 1}

→ **Suspicious statement: {r= i - j}**

LocFaults – Exemple (cont.)





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

Change decision for 2d **IF**: Post-condition **holds**

CSP: $k0 = 0 \wedge k1 = k0 + 2 \wedge \neg((k1 = 1 \& I \neq j))$

Potential corrections: {k0 =0},{k=k+2},
{k=1&I ≠ j}

Computing all MCS(Minimal Correction Set)

Liffiton & Sakallah-2007

All_MCSes(ϕ)

1. $\phi' \leftarrow$ AddYVars(ϕ) *% Adds y_i selector variables*
2. MCSes $\leftarrow \varnothing$
3. k $\leftarrow$ 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 $\leftarrow$ IncrementalSAT(ϕ'_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 – experiments

Benchs	CE	E	Locfaults				BugAssist	
			0		1			
V7	i=2,j=1, k=2	58	**58**	0,77 s	{31},{37}, {27},{32}	**0,86 s**	{72, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62}	**20,48 s**
V8	i=3,j=4, k=3	61	**61**	0,74 s	{29},{35}, {30},{25}	**0,88 s**	{19, **61**, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 44}	**25,72 s**

BugAssist: global approach based on MaxSat, merges the complements of ***MaxSat*** in a single set of suspicious statements

V7 and **V8** : variations of ***Tritype***

Input: three ***positive integers***, the triangle sides

Output: type of triangle (isosceles, equilateral, scalene, not a triangle)

V7 returns the ***product of the 3 sides***

V8 computes **the *square of the surface of the triangle by using Heron's formula***

LocFaults – Sum up

- **Flow-based** and **incremental** approach
  $\longrightarrow$ locates ***suspicious statements around the path*** of the counter-example

- **Constraint-based framework**
  - well adapted for handling arithmetic operations … on ***integers***
  - can be extended for handling programs with ***floating-point numbers*** computations (**?**)

Conclusion

- **BMC (Bounded Model Checking)**
  - Goal : ***Finding counter-examples violating an assertion***
  - Contribution of CP: ***Various solvers*** and ***search strategies***
  - Limit of CP: ***efficient pruning is a critical issue***

- **Program analysis**
  - Goal: ***Get rid of false alarms***
  - Contribution of CP : ***Refining abstraction, suspicious values***
  - Limit of CP: ***high computation cost***

- **Fault localization**
  - Goal: ***locations of potentially faulty statements***
  - Contribution of CP : **flow-based & incremental approach**
  - Limit of CP: **no global view**