Localisation d'erreurs à base de contraintes

LocFaults: Une nouvelle approche basée sur les contraintes et dirigée par les flots pour la localisation d'erreurs

Mohammed Bekkouche, Michel Rueher, Hélène Collavizza

{bekkouche,helen,rueher}@unice.fr

I3S/CNRS, BP 121, 06903 Sophia Antipolis Cedex, France
Université de Nice-Sophia Antipolis

SAC 2015
13 au 17 avril 2015

Plan

Introduction
Motivation

L'aide à localisation d'erreurs est une tâche importante pour débugger un programme erroné mais complexe en même temps

→ Lorsqu'un programme est non-conforme vis-à-vis de sa spécification, à savoir, le programme est erroné :

- Les outils de BMC(Bounded Model Checking) et de test peuvent générer un ou plusieurs contre-exemples
- **La trace du contre-exemple** est souvent longue et compliquée à comprendre
- L'identification des parties erronées du code est difficile même pour les programmeurs expérimentés

Introduction

Le problème : entrées et objectif

Entrées

- Un programme en contradiction avec sa spécification
- La postcondition violée POST
- Un contre-exemple CE fourni par un outil BMC

Objectif

Un ensemble réduit d'instructions suspectes permettant au programmeur de comprendre l'**origine de ses erreurs**

Introduction
Les idées

1. Le programme est modélisé en un CFG en forme DSA
2. Le programme et sa spécification sont traduits en contraintes numériques
3. *CE* : un contre-exemple, PATH : **un chemin erroné**
4. Le CSP $C = CE \cup PATH \cup POST$ **est inconsistant**

Les questions clés

- Quelles sont les instructions erronées dans *PATH* qui rendent C **inconsistant** ?
- Quels sous-ensembles enlever pour restaurer **la faisabilité** dans C ?
- Quels **chemins** explorer ?

Exemple

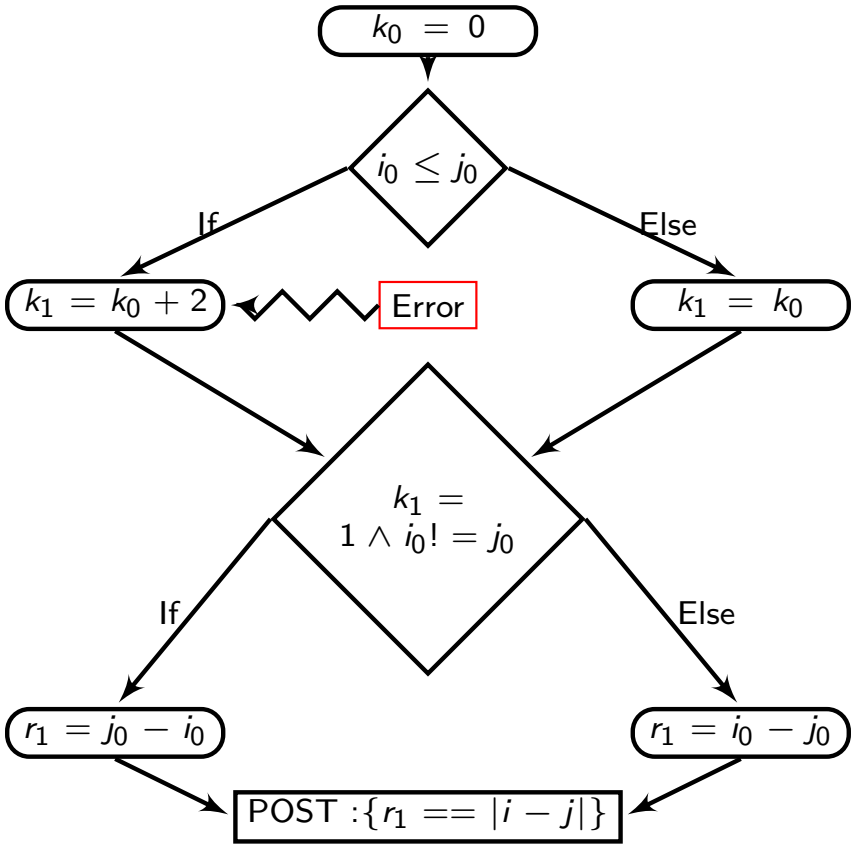
Calcul de la valeur absolue de i-j

```
1  class AbsMinus {
2  /*returns|i-j|,the absolute value of i minus j*/
3  /*@ ensures
4    @ (result==|i-j|);
5    @*/
6    void 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     }
12     if (k == 1 && i != j) {
13         result = j-i;
14     }
15     else {
16         result = i-j;
17     }
18   }
19 }
```

Exemple

Calcul de la valeur absolue de i-j

```
1  class AbsMinus {
2  /*returns|i-j|,the absolute value of i minus j*/
3  /*@ ensures
4    @ (result==|i-j|);
5    @*/
6    void 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     }
12     if (k == 1 && i != j) {
13         result = j-i;
14     }
15     else {
16         result = i-j;
17     }
18   }
19 }
```



Exemple

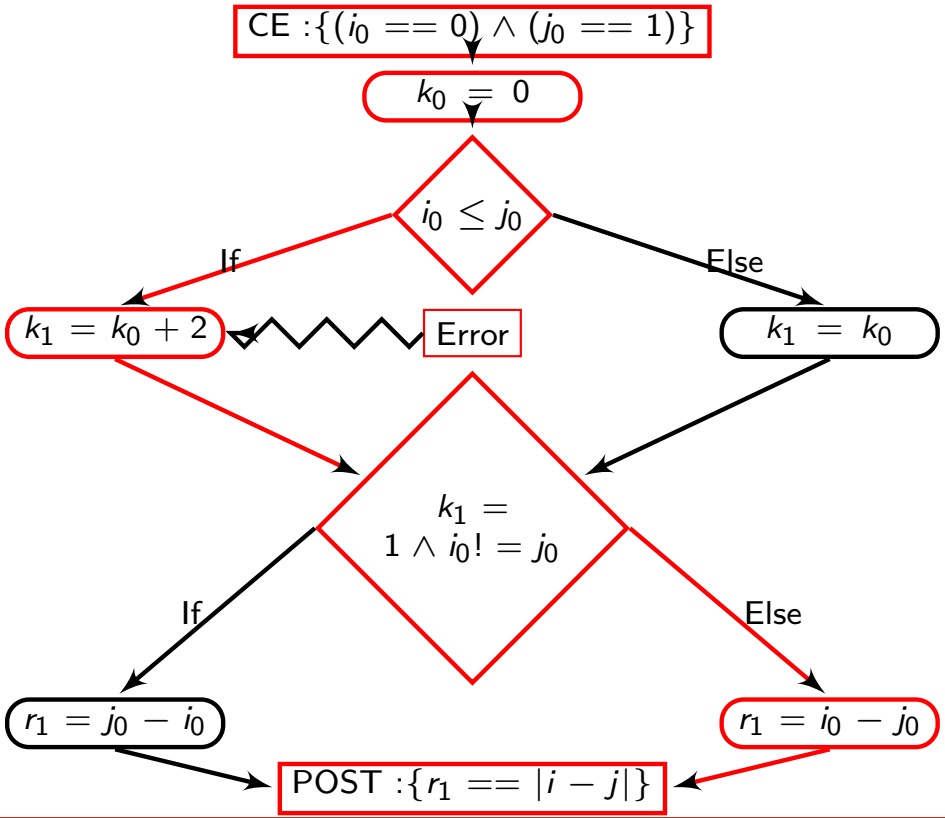
Le **chemin** du contre-exemple

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

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

Seulement un seul MCS sur le chemin :

$\{r_1 = i_0 - j_0\}$



Exemple

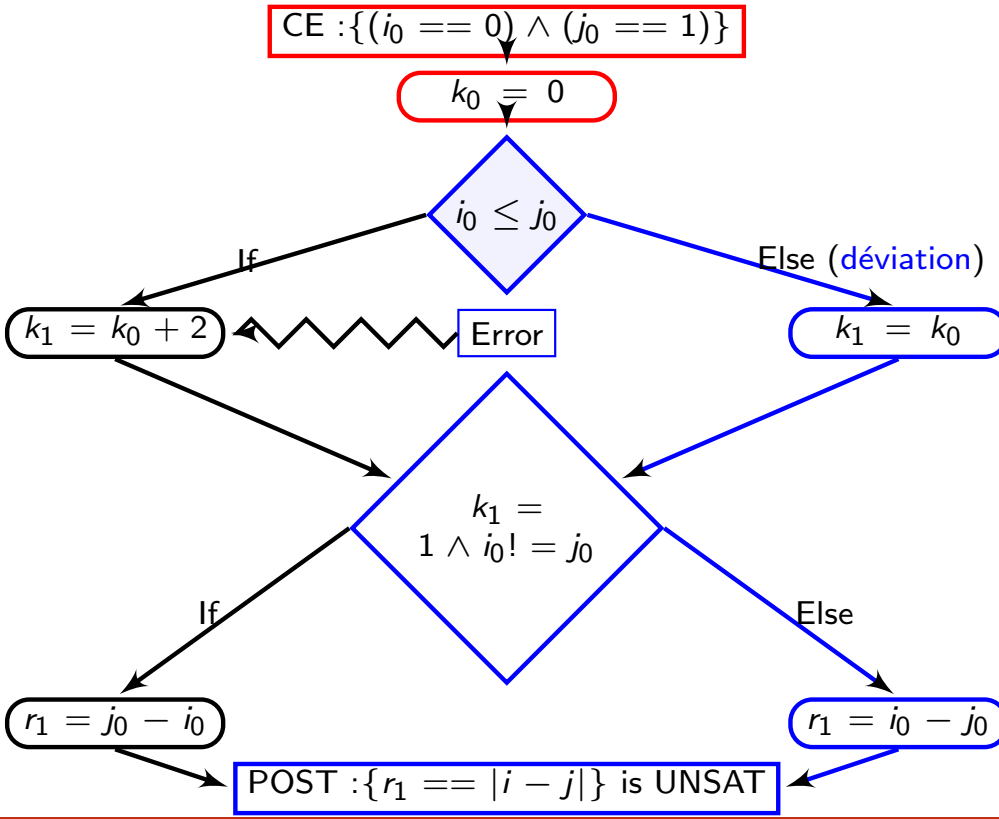
Le **chemin** obtenu en **déviant** la condition $i_0 \leq j_0$

La condition **déviée** : $\{i_0 \leq j_0\}$

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

$P \cup \{r_1 = |i - j|\}$ **est inconsistant**

La déviation $\{i_0 \leq j_0\}$ **ne corrige pas le programme**



Exemple

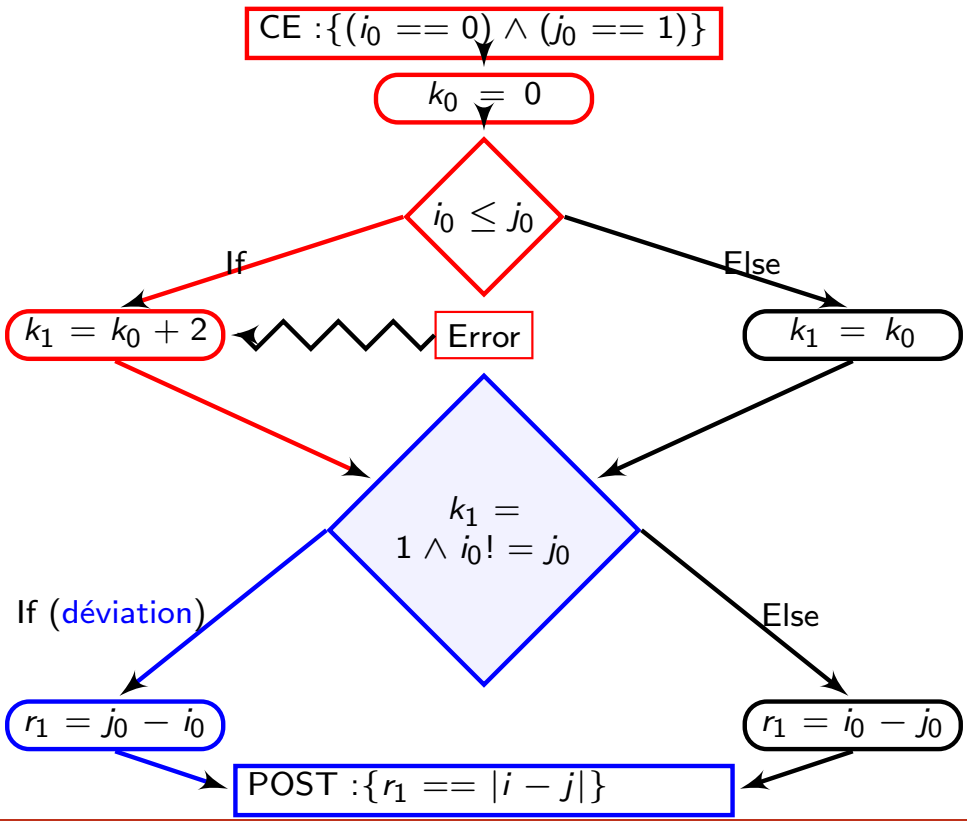
Le **chemin** obtenu en **déviant** la condition $k_1 = 1 \wedge i_0 ! = j_0$

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

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

La déviation $\{(k_1 = 1 \wedge i_0 ! = j_0)\}$ **corrige** le programme

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



Exemple

Le **chemin** obtenu en **déviant** la condition $k_1 = 1 \wedge i_0 ! = j_0$

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

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

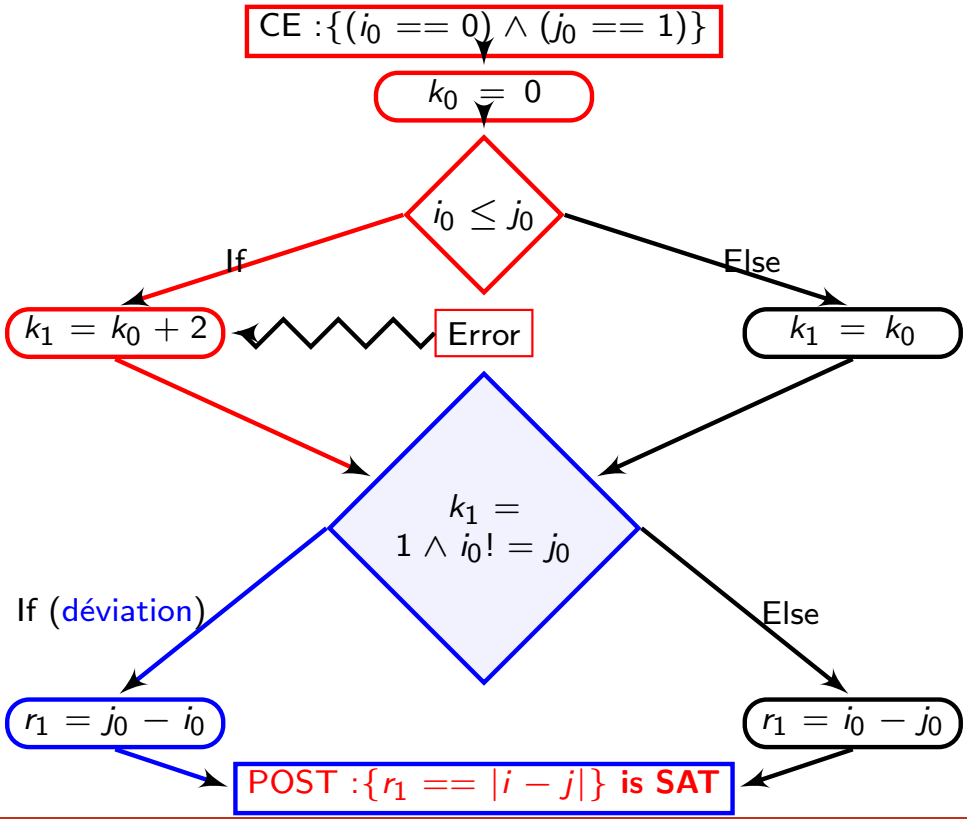
$P \cup \{r_1 = |i - j|\}$ **est inconsistant**

La déviation $\{(k_1 = 1 \wedge i_0 ! = j_0)\}$ **corrige** le programme

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

C **est inconsistant**

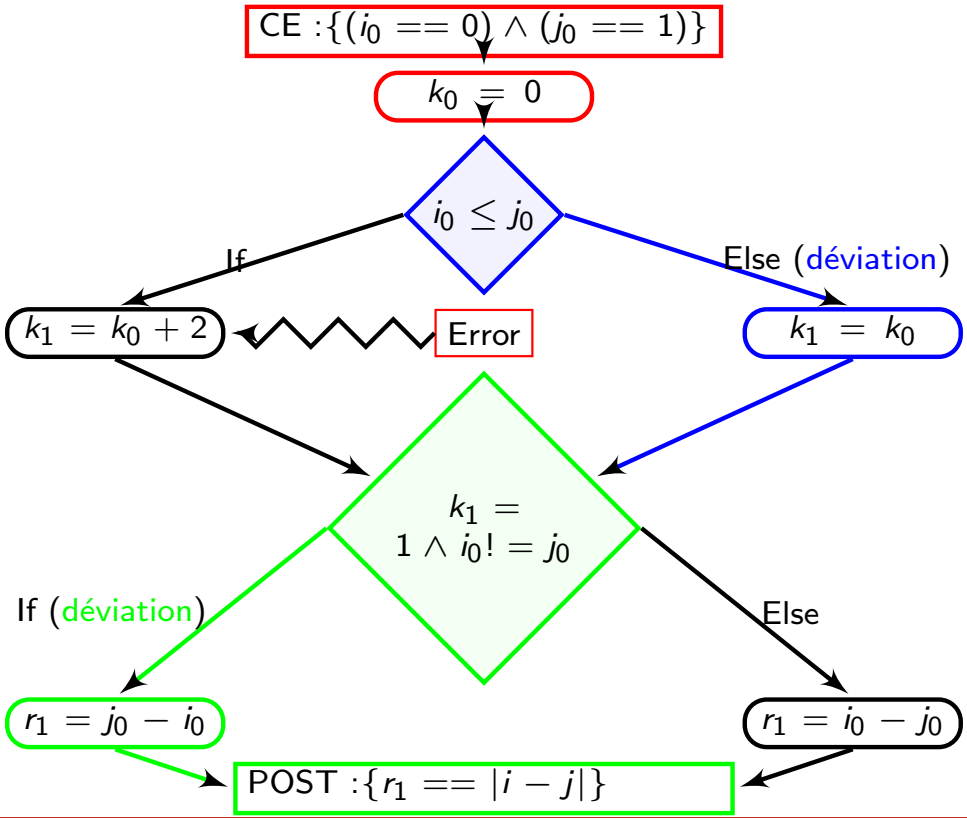
MCS sur le chemin : $\{k_0 = 0\}$, $\{k_1 = k_0 + 2\}$



Exemple

Le **chemin** d'une **déviation non-minimale** : $\{i_0 \leq j_0, k_1 = 1 \wedge i_0! = j_0\}$

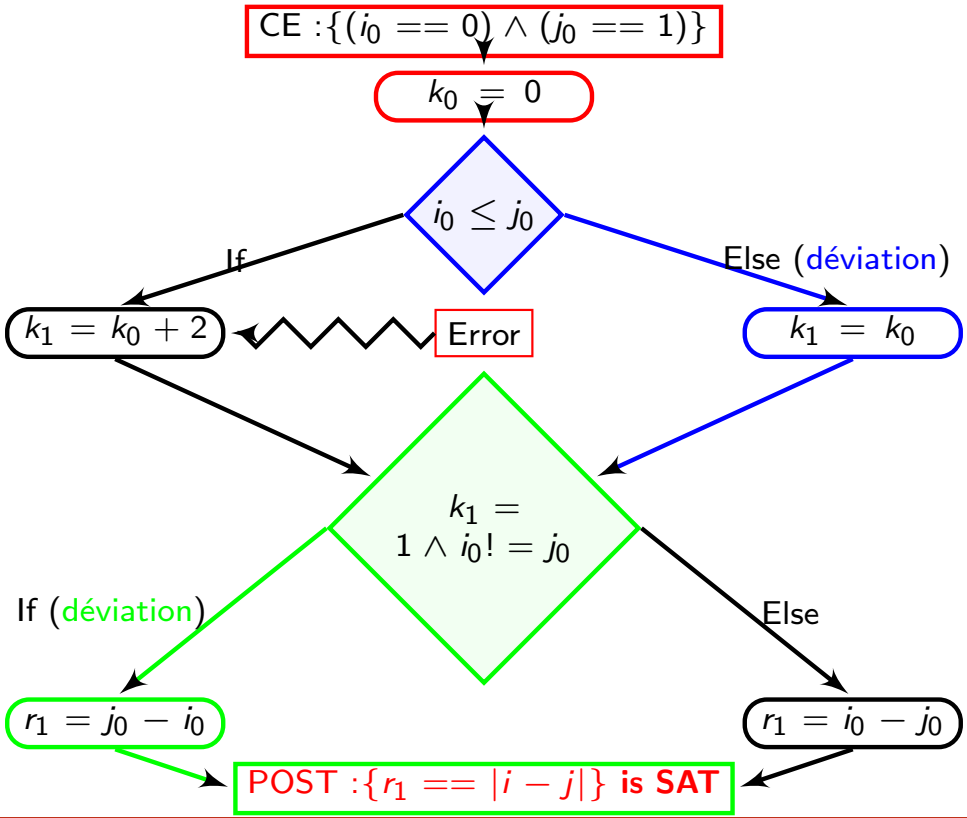
Les conditions **déviées** :
$\{i_0 \leq j_0, (k_1 = 1 \wedge i_0! = j_0)\}$
$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\}$



Exemple

Le **chemin** d'une **déviation non-minimale** : $\{i_0 \leq j_0, k_1 = 1 \wedge i_0! = j_0\}$

Les conditions **déviées** :
$\{i_0 \leq j_0, (k_1 = 1 \wedge i_0! = j_0)\}$
$P = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\}$
$P \cup \{r_1 = |i - j|\}$ **est consistant**
La déviation est non-minimale



L'approche LocFaults

MCS : Minimal Correction Subset

MCS : Définition

Soit C un ensemble **infaisable** de contraintes

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

L'approche LocFaults

MCS : Minimal Correction Subset

MCS : Définition

Soit C un ensemble **infaisable** de contraintes

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

L'approche LocFaults

MCS : Minimal Correction Subset

MCS : Définition

Soit C un ensemble **infaisable** de contraintes

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

L'approche LocFaults

MCS : Minimal Correction Subset

MCS : Définition

Soit C un ensemble **infaisable** de contraintes

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

MCS : Exemple

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

L'approche LocFaults

L'algorithme (LocFaults)

- Calcul des **MCS** sur le **chemin** du CE
- Exploration DFS du CFG en propageant le CE et **en déviant** au moins k instructions conditionnelles $c_1, .., c_k$
  - P : contraintes de propagation issues du CE (de la forme $variable = constante$)
  - C : contraintes du **chemin** jusqu'à c_k
  - Si $\underline{P \models POST}$:
    * $\{\neg c_1, .., \neg c_k\}$ est une **correction**,
    * **MCS** de $C \cup \{\neg c_1, .., \neg c_k\}$ sont des **corrections**
- Une borne pour les **MCS** calculés et les conditions **déviées**

Expérience pratique
Outils utilisés

- **LocFaults** : notre implémentation
  → Les solveurs CP OPTIMIZER et CPLEX d'IBM
  → L'outil CPBPV pour générer le CFG et CE
  → Benchmarks : les programmes Java
- **BugAssist** : l'outil de localisation d'erreurs implémentant l'approche `BugAssist`
  → Le solveur MaxSAT MSUnCore2
  → Benchmarks : les programmes ANSI-C

Expérience expérimentale
Les programmes utilisés

- Une variation sur le programme **Tritype** :
  → TritypeV1, TritypeV2,TritypeV3, TritypeV4, TritypeV5
  → TritypeV6 (renvoie **le périmètre** d'un triangle)
  → TritypeV7, TritypeV8 (renvoient **des expressions non-linéaires**)
- **TCAS(Traffic Collision Avoidance System)**, un benchmark réaliste :
  → 1608 cas de tests, sauf les cas qui débordent le tableau *PositiveRAAltThresh*
  → TcasKO ... TcasKO41

Expérience pratique

Résultats (**MCS** identifiés)

Programme	Contre-exemple	Erreurs	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` fournit une localisation plus explicative et informative

Expérience pratique

Résultats (temps de calcul pour les programmes non linéaires)

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` est plus rapide que `BugAssist` pour ces benchmarks

Expérience pratique

Résultats (nombre d'erreurs localisées pour 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

Les performances de `LocFaults` et `BugAssist` sont très similaires sur ces programmes bien adaptés pour un solveur booléen

Travaux connexes

Approche basées sur SAT

BugAssist

- Une méthodes BMC, comme la notre
- **Les principales différences :**

  → Elle transforme le programme en entier en une formule SAT

  → Elle se base sur l'utilisation des solveurs MaxSAT

\+ Une approche globale

− Elle n'est pas efficace pour les programmes avec calcul numérique

Travaux connexes

Approches basées sur le test systématique

Tarantula, Ochiai, AMPLE, Jaccard, Heuristics III

- Classement des instructions suspectes détectées durant l'exécution d'une batterie de tests

\+ Des approches simples

\− Besoins de **beaucoup** de cas de tests
Les approches qui nécessitent l'existence d'un oracle
→ Décidez si le résultat de **dizaines de milliers** de tests est **correct**

Notre cadre moins exigeant

→ Bounded Model Checking

Conclusion et perspectives

- Notre approche incrémentale basée sur les flots est une bonne manière pour aider le programmeur à la chasse aux bugs
  → Elle localise les erreurs autour du chemin du contre-exemple
- Nous prévoyons :
  - de développer une version interactive de notre outil :
  → pour fournir les localisations l'une après l'autre
  → pour bénéficier des connaissances de l'utilisateur pour sélectionner la condition qui doit être déviée
  - d'étendre notre approche de manière simple pour la localisation des erreurs dans les programmes avec calcul sur flottants