Soutenance de Thèse de Doctorat :

Combinaison des techniques de Bounded Model Checking et de Programmation Par Contraintes pour l'aide à la localisation d'erreurs

Mohammed Bekkouche

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

10 décembre 2015

Plan

Introduction

Motivation de la Thèse : Conséquences de bugs

- Les bugs informatiques peuvent avoir de terribles conséquences, exemples :

Ariane 5 Flight 501 (1996)

LA Airport Flights Grounded (2007)

Soviet Gas Pipeline Explosion (1982)

Black Monday (1987)

Introduction

Motivation de la Thèse : Conséquences de bugs

- Les bugs informatiques peuvent avoir de terribles conséquences, exemples :

Ariane 5 Flight 501 (1996)

LA Airport Flights Grounded (2007)

Soviet Gas Pipeline Explosion (1982)

Black Monday (1987)

Introduction

Motivation de la Thèse : Le débogage de programmes

- Le processus de débogage est **essentiel** mais **coûteux**
- Débogage : localisation et correction des erreurs
- La motivation principale de la localisation d'erreurs est de diminuer le coût de la correction des bugs

Dans le développement de logiciels :

1. Le débogage peut être l'activité la plus difficile
2. La localisation d'erreurs est l'étape qui coûte le plus sur les aspects de débogage

Cette thèse propose une nouvelle approche basée sur la Programmation Par Contraintes pour l'aide à la localisation d'erreurs à partir d'un cas d'erreur (un contre-exemple)

Introduction

Motivation de la Thèse : Le débogage de programmes

- Le processus de débogage est **essentiel** mais **coûteux**
- Débogage : localisation et correction des erreurs
- La motivation principale de la localisation d'erreurs est de diminuer le coût de la correction des bugs

Dans le développement de logiciels :

1. Le débogage peut être l'activité la plus difficile
2. La localisation d'erreurs est l'étape qui coûte le plus sur les aspects de débogage

Cette thèse propose une nouvelle approche basée sur la Programmation Par Contraintes pour l'aide à la localisation d'erreurs à partir d'un cas d'erreur (un contre-exemple)

Introduction

Motivation de la Thèse : Le débogage de programmes

- Le processus de débogage est **essentiel** mais **coûteux**
- Débogage : localisation et correction des erreurs
- La motivation principale de la localisation d'erreurs est de diminuer le coût de la correction des bugs

Dans le développement de logiciels :

1. Le débogage peut être l'activité la plus difficile
2. La localisation d'erreurs est l'étape qui coûte le plus sur les aspects de débogage

Cette thèse propose une nouvelle approche basée sur la Programmation Par Contraintes pour l'aide à la localisation d'erreurs à partir d'un cas d'erreur (un contre-exemple)

Introduction

Problématique de la localisation d'erreurs

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

Aperçu de notre solution : Entrées & Sortie

Entrées

- Un programme non conforme vis-à-vis de sa spécification : la postcondition POST violée
- Un contre-exemple CE fourni par un outil BMC

Sorties

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

Introduction

Aperçu de notre solution : 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 ?

Introduction

Aperçu de notre solution : **MCS**(**M**inimal **C**orrection **S**ubset)

MCS : Définition

Soit C un ensemble de **contraintes infaisable**

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

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

Introduction

Aperçu de notre solution : MCS(Minimal Correction Subset)

MCS : Définition

Soit C un ensemble de **contraintes infaisable**

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

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

Introduction

Aperçu de notre solution : **MCS**(**M**inimal **C**orrection **S**ubset)

MCS : Définition

Soit C un ensemble de **contraintes infaisable**

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

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

Introduction

Aperçu de notre solution : **MCS**(**M**inimal **C**orrection **S**ubset)

MCS : Définition

Soit C un ensemble de **contraintes infaisable**

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

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

Introduction

Aperçu de notre solution : Exemple explicatif

Calcul de la valeur absolue de i-j :

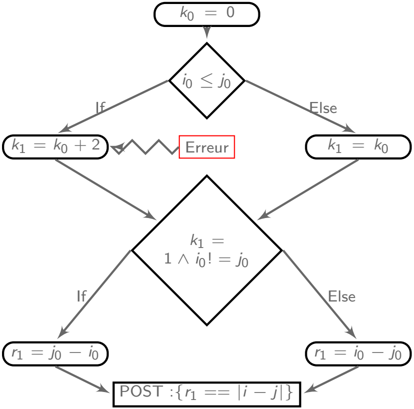
```java
class AbsMinus {
/*returns|i−j|,the absolute value of i minus j*/
/*@ ensures
  @ (result==|i−j|);
  @*/
  void AbsMinus (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
       k = k+2; //error:k = k+2 instead of k=k+1
    }
    if (k == 1 && i != j) {
        result = j−i;
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

Introduction

Aperçu de notre solution : Exemple explicatif

Calcul de la valeur absolue de 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;
    }
  }
}
```



Introduction

Aperçu de notre solution : Exemple explicatif

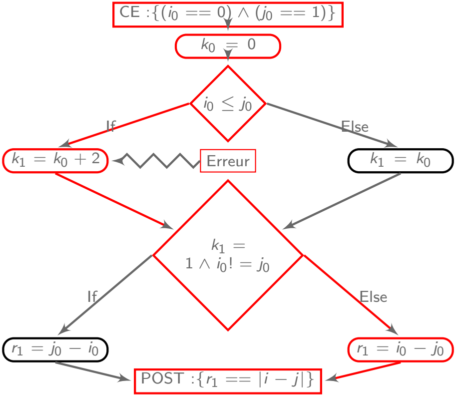
Le chemin du contre-exemple :

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

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

Seulement un seul MCS sur le chemin :

$\{r_1 = i_0 - j_0\}$



Introduction

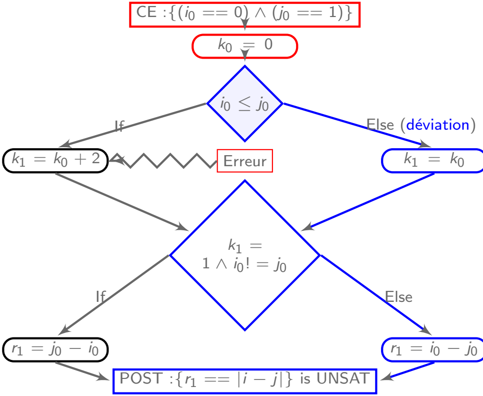
Aperçu de notre solution : Exemple explicatif

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

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

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

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



Introduction

Aperçu de notre solution : Exemple explicatif

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

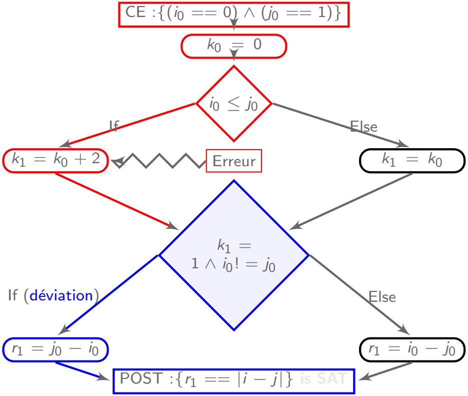
La condition **déviée** : $\{(k_1 = 1 \wedge i_0! = j_0)\}$

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

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

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

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



Introduction

Aperçu de notre solution : Exemple explicatif

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

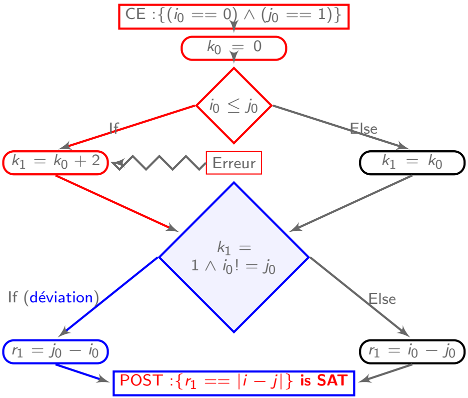
La condition **déviée** : $\{(k_1 = 1 \wedge i_0 != j_0)\}$

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

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

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

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



Introduction

Principales contributions

Localiser les erreurs dans un programme incorrect → Diagnostiquer un CSP infaisable

- Le CSP représente les contraintes du CE, du programme et de l'assertion violée
- L'ensemble calculé peut être un ensemble de correction minimal (MCS) ou ensemble inconsistant minimal (MUS)
- Expliquer l'inconsistance dans un CSP est difficile
- `BugAssist` est une méthode de localisation d'erreurs qui calcule la fusion des MCSs avec un solveur MaxSAT
- Elle devient inefficace pour les programmes qui contiennent des opérations arithmétiques sur les entiers
- `LocFaults` utilise des solveur de contraintes pour calculer les MCSs sur les chemins incorrects du programme

Introduction

Principales contributions : `LocFaults` vs. `BugAssist`

* Nous ne transformons pas la totalité du programme en un système de contraintes
* Nous calculons les MCSs **uniquement** sur le chemin du contre-exemple et les chemins déviés du contre-exemple permettant de satisfaire la postcondition
* Nous traduisons les instructions en contraintes numériques
* Nous utilisons un algorithme générique pour calculer les MCSs
* Nous bornons la taille des MCSs générés et le nombre de conditions déviées.
* Nous pouvons faire collaborer plusieurs solveurs durant le processus de localisation

Notre approche

DCM : Déviation de Correction Minimale

DCM : Définition

Soit un programme erroné modélisé en un CFG $G = (C, A, E)$: C est l'ensemble des nœuds conditionnels, A est l'ensemble des blocs d'affectation, E est l'ensemble des arcs, et un contre-exemple.

Une DC (*Déviation de Correction*) est un ensemble $D \subseteq C$ telle que la propagation du contre-exemple sur l'ensemble des instructions de G à partir de la racine, tout en ayant nié chaque condition dans D, permet de satisfaire la postcondition

$$D \subseteq C \text{ est une } \mathbf{DCM} \Leftrightarrow \begin{cases} D \text{ est une DC} \\ \nexists C' \subset D : C' \text{ est une DC} \end{cases}$$

Notation $\leq k$-DCM

Le problème $\leq k$-DCM consiste à trouver toutes les DCMs de taille inférieure ou égale à k

Notre approche

DCM : Déviation de Correction Minimale

DCM : Définition

Soit un programme erroné modélisé en un CFG $G = (C, A, E)$: C est l'ensemble des nœuds conditionnels, A est l'ensemble des blocs d'affectation, E est l'ensemble des arcs, et un contre-exemple.

Une DC (*Déviation de Correction*) est un ensemble $D \subseteq C$ telle que la propagation du contre-exemple sur l'ensemble des instructions de G à partir de la racine, tout en ayant nié chaque condition dans D, permet de satisfaire la postcondition

$$D \subseteq C \text{ est une } \mathbf{DCM} \Leftrightarrow \begin{cases} D \text{ est une DC} \\ \nexists C' \subset D : C' \text{ est une DC} \end{cases}$$

Notation $\leq k$-DCM

Le problème $\leq k$-DCM consiste à trouver toutes les DCMs de taille inférieure ou égale à k

Notre approche

DCM : Déviation de Correction Minimale

DCM : Définition

Soit un programme erroné modélisé en un CFG $G = (C, A, E)$: C est l'ensemble des nœuds conditionnels, A est l'ensemble des blocs d'affectation, E est l'ensemble des arcs, et un contre-exemple.

Une DC (*Déviation de Correction*) est un ensemble $D \subseteq C$ telle que la propagation du contre-exemple sur l'ensemble des instructions de G à partir de la racine, tout en ayant nié chaque condition dans D, permet de satisfaire la postcondition

$$D \subseteq C \text{ est une } \textbf{DCM} \Leftrightarrow \begin{cases} D \text{ est une DC} \\ \nexists C' \subset D : C' \text{ est une DC} \end{cases}$$

Notation $\leq k$-DCM

Le problème $\leq k$-DCM consiste à trouver toutes les DCMs de taille inférieure ou égale à k

Notre approche

DCM : Déviation de Correction Minimale

DCM : Définition

Soit un programme erroné modélisé en un CFG $G = (C, A, E)$: C est l'ensemble des nœuds conditionnels, A est l'ensemble des blocs d'affectation, E est l'ensemble des arcs, et un contre-exemple.

Une DC (*Déviation de Correction*) est un ensemble $D \subseteq C$ telle que la propagation du contre-exemple sur l'ensemble des instructions de G à partir de la racine, tout en ayant nié chaque condition dans D, permet de satisfaire la postcondition

$$D \subseteq C \text{ est une } \textbf{DCM} \Leftrightarrow \begin{cases} D \text{ est une DC} \\ \nexists C' \subset D : C' \text{ est une DC} \end{cases}$$

Notation $\leq k$-DCM

Le problème $\leq k$-DCM consiste à trouver toutes les DCMs de taille inférieure ou égale à k

Notre approche

Déviation de Correction non-Minimale

Le **chemin** d'une déviation non-minimale

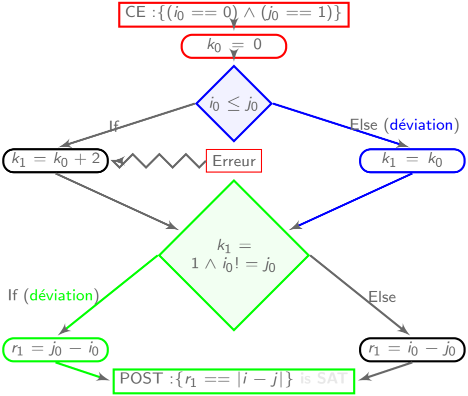
$\{i_0 \leq j_0, k_1 = 1 \wedge i_0 != j_0\}$:

Les conditions **déviées** :

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

$\{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = 0, r_1 = 1\} \cup \{r_1 = |i - j|\}$ **est consistant**

La déviation est non-minimale

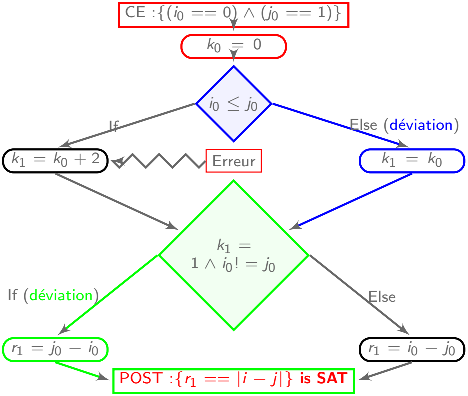


Notre approche

Déviation de Correction non-Minimale

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

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



Notre approche

LocFaults : BMC à base de contraintes (CPBPV)

- Contexte : Programmes avec des opérations numériques sur les entiers ou nombres à virgule flottante
- Objectif : Trouver des contre-exemples violant une assertion

BMC sur la base de la programmation par contraintes

- Le programme est déplié b fois
- Le programme déplié (et simplifié) sous forme SSA/DSA et la négation de la propriété sont traduits à la volée en un système de contrainte $Cs = PRE \wedge PROG_b \wedge \neg POST$
  $\rightarrow$ Cs est satisfiable pour un chemin complet, ssi il existe un contre-exemple de profondeur inférieure à b
- Divers solveurs et stratégies peuvent être utilisés

Notre approche

LocFaults : MCSs pour localiser les erreurs

- Localisation des erreurs détectées par la phase de BMC
- CE une instanciation des variables qui satisfait $PRE \cup PATH \cup \neg POST$ (CE est un contre-exemple) $\rightarrow C = CE \cup PRE \cup PATH \cup POST$ est inconsistant

Démarche de LocFaults pour localiser les erreurs

1. L'erreur peut se trouver dans une affectation sur le chemin du CE : LocFaults calcule les MCSs de C pour localiser les erreurs sur le chemin du CE
2. L'erreur peut provenir d'un mauvais branchement : LocFaults calcule les MCSs des CSP obtenus en déviant des branchements par rapport au comportement induit par CE

Notre approche

LocFaults : Algorithme

- **Entrées :** $PROG_b$, CE, b_{cond} : une borne sur le nombre de conditions qui sont déviées, b_{mcs} : une borne sur le nombre de MCSs générés.
- **Sorties :** Une liste de corrections possibles

1. Construction du CFG du $PROG_b$
2. Exploration du CFG : `LocFaults` appelle la fonction `DFS` sur le chemin du CE (i.e. en déviant 0 condition) puis en acceptant au plus b_{cond} déviations
   - Explorer le chemin du contre-exemple et les chemins avec au plus b_{cond} déviations
   - Calculer les ensembles MCSs avec au plus b_{mcs} instructions suspectes

Notre approche

LocFaults : Algorithme

1. LocFaults calcule au plus b_{mcs} MCSs sur le CSP du chemin du CE ($b_{cond} = 0$)
2. LocFaults essaye de dévier une condition ($b_{cond} = 1$). Si le CSP du chemin dévié satisfait la postcondition, il y a deux types d'ensemble d'instructions suspectes :
   - La condition *cond* est source de l'erreur
   - LocFaults calcule au plus b_{mcs} MCSs sur le CSP des contraintes collectées sur le chemin qui arrivent à *cond*
3. Le processus (2) est répété sur chaque nœud conditionnel du chemin du CE
4. **Processus pour** $b_{cond} > 1$
   - Un noeud conditionnel **n** est marqué avec le nombre de déviations sur le chemin courant avant d'atteindre **n**
   - À l'étape b_{cond}, une décision pour un noeud marqué avec b'_{cond} est seulement déviée **ssi** $b'_{cond} < b_{cond}$

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})
Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}
début
    $C' \leftarrow$ AddYVars(C) ; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
    tant que $\mathrm{SAT}(C') \wedge k \leq b_{mcs}$ **faire**
        $C'_k \leftarrow C' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$,$k$)
        tant que $\mathrm{SAT}(C'_k)$ **faire**
            $newMCS \leftarrow \emptyset$
            pour chaque indicateur y_i **faire**
                Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.
                si $val(y_i) = 0$ **alors**
                    $newMCS \leftarrow newMCS \cup \{c_i\}$.
                fin
            fin
            $MCS.add(newMCS)$.
            $C'_k \leftarrow C'_k \wedge$ BlockingClause($newMCS$)
            $C' \leftarrow C' \wedge$ BlockingClause($newMCS$)
        fin
        $k \leftarrow k + 1$
    fin
    retourner MCS
fin

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})
Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}
début
 $C' \leftarrow$ AddYVars(C) ; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
 tant que $\underline{\mathrm{SAT}(C') \wedge k \leq b_{mcs}}$ **faire**
  $C'_k \leftarrow C' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$,$k$)
  tant que $\underline{\mathrm{SAT}(C'_k)}$ **faire**
   $newMCS \leftarrow \emptyset$
   pour chaque $\underline{\text{indicateur } y_i}$ **faire**
    Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.
    si $\underline{val(y_i) = 0}$ **alors**
     $newMCS \leftarrow newMCS \cup \{c_i\}$.
    fin
   fin
   $MCS.add(newMCS)$.
   $C'_k \leftarrow C'_k \wedge$ BlockingClause($newMCS$)
   $C' \leftarrow C' \wedge$ BlockingClause($newMCS$)
  fin
  $k \leftarrow k + 1$
 fin
 retourner MCS
fin

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})
Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}
début
    $C' \leftarrow$ AddYVars(C) ; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
    tant que $\underline{\mathrm{SAT}(C') \wedge k \leq b_{mcs}}$ **faire**
        $C'_k \leftarrow C' \wedge$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$,$k$)
        tant que $\underline{\mathrm{SAT}(C'_k)}$ **faire**
            $newMCS \leftarrow \emptyset$
            pour chaque indicateur y_i **faire**
                Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.
                si $\underline{val(y_i) = 0}$ **alors**
                    $newMCS \leftarrow newMCS \cup \{c_i\}$.
                fin
            fin
            $MCS.add(newMCS)$.
            $C'_k \leftarrow C'_k \wedge$ BlockingClause($newMCS$)
            $C' \leftarrow C' \wedge$ BlockingClause($newMCS$)
        fin
        $k \leftarrow k + 1$
    fin
    retourner MCS
fin

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

```
Fonction MCS(C, b_mcs)
Entrées: C : Ensemble de contraintes infaisable, b_mcs : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_mcs
début
    C' ← AddYVars(C) ; MCS ← ∅ ; k ← 1 ;
    tant que SAT(C') ∧ k ≤ b_mcs faire
        C'_k ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n}, k)
        tant que SAT(C'_k) faire
            newMCS ← ∅
            pour chaque indicateur y_i faire
                Soit y_i l'indicateur de la contrainte c_i ∈ C, et val(y_i) la valeur de y_i dans la
                solution calculée de C'_k.
                si val(y_i) = 0 alors
                    newMCS ← newMCS ∪ {c_i}.
                fin
            fin
            MCS.add(newMCS).
            C'_k ← C'_k ∧ BlockingClause(newMCS)
            C' ← C' ∧ BlockingClause(newMCS)
        fin
        k ← k + 1
    fin
    retourner MCS
fin
```

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})
Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}
début
  $C' \leftarrow$ AddYVars(C) ; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
  tant que $\mathrm{SAT}(C') \land k \leq b_{mcs}$ **faire**
    $C'_k \leftarrow C' \land$ AtMost($\{\neg y_1, \neg y_2, \ldots, \neg y_n\}$,$k$)
    tant que $\mathrm{SAT}(C'_k)$ **faire**
      $newMCS \leftarrow \emptyset$
      pour chaque indicateur y_i **faire**
        Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.
        si $val(y_i) = 0$ **alors**
          $newMCS \leftarrow newMCS \cup \{c_i\}$.
        fin
      fin
      $MCS.add(newMCS)$.
      $C'_k \leftarrow C'_k \land$ BlockingClause($newMCS$)
      $C' \leftarrow C' \land$ BlockingClause($newMCS$)
    fin
    $k \leftarrow k + 1$
  fin
  retourner MCS
fin

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})
Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier
Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}
début
  $C' \leftarrow \textsc{AddYVars}(C)$; $MCS \leftarrow \emptyset$; $k \leftarrow 1$;
  tant que $\mathrm{SAT}(C') \wedge k \leq b_{mcs}$ **faire**
    $C'_k \leftarrow C' \wedge \textsc{AtMost}(\{\neg y_1, \neg y_2, \ldots, \neg y_n\}, k)$
    tant que $\mathrm{SAT}(C'_k)$ **faire**
      $newMCS \leftarrow \emptyset$
      pour chaque indicateur y_i **faire**
        Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.
        si $val(y_i) = 0$ **alors**
          $newMCS \leftarrow newMCS \cup \{c_i\}$.
        fin
      fin
      $MCS.add(newMCS)$.
      $C'_k \leftarrow C'_k \wedge \textsc{BlockingClause}(newMCS)$
      $C' \leftarrow C' \wedge \textsc{BlockingClause}(newMCS)$
    fin
    $k \leftarrow k + 1$
  fin
  retourner MCS
fin

Notre approche

LocFaults : Calcul MCSs dans le CSP construit

Algorithme de **Liffiton** et **Sakallah** pour calculer les MCSs :

Fonction MCS(C,b_{mcs})

Entrées: C : Ensemble de contraintes infaisable, b_{mcs} : Entier

Sorties: MCS : Liste de MCS de C de cardinalité inférieure à b_{mcs}

début

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

  tant que $\text{SAT}(C') \wedge k \leq b_{mcs}$ **faire**

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

    tant que $\text{SAT}(C'_k)$ **faire**

      $newMCS \leftarrow \emptyset$

      pour chaque indicateur y_i **faire**

        Soit y_i l'indicateur de la contrainte $c_i \in C$, et $val(y_i)$ la valeur de y_i dans la solution calculée de C'_k.

        si $val(y_i) = 0$ **alors**

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

        fin

      fin

      $MCS.add(newMCS)$.

      $C'_k \leftarrow C'_k \wedge$ BlockingClause($newMCS$)

      $C' \leftarrow C' \wedge$ BlockingClause($newMCS$)

    fin

    $k \leftarrow k + 1$

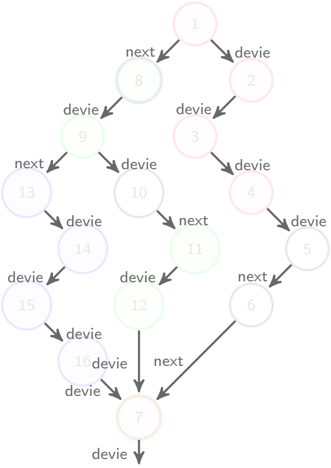
  fin

  retourner MCS

fin

Notre approche

LocFaults : Illustration du déroulement du marquage des noeuds

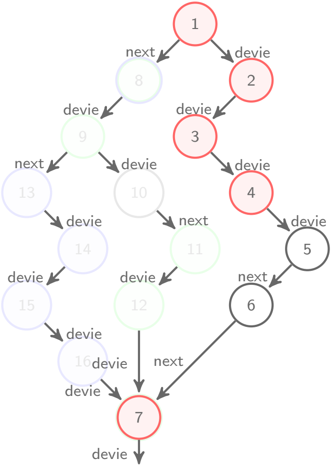


le chemin $< 1, 2, 3, 4, 5, 6, 7, \ldots, POST >$ est correct
le chemin $< 1, 8, 9, 10, 11, 12, 7, \ldots, POST >$ est correct

- À l'étape $k = 5$, notre algorithme a identifié deux déviations de correction minimales de taille égale à 5 :
  1. $D_1 = \{1, 2, 3, 4, 7\}$, le nœud "7" est marqué par la valeur 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, elle a été autorisée, car la valeur de la marque du nœud "7" est égale à la cardinalité de D_2.
- À l'étape $k = 6$, l'algorithme a suspendu la déviation suivante $D_3 = \{8, 13, 14, 15, 16, 7\}$, car la cardinalité de D_3 est supérieure strictement à la valeur de l'étiquette du nœud "7"

Notre approche

LocFaults : Illustration du déroulement du marquage des noeuds



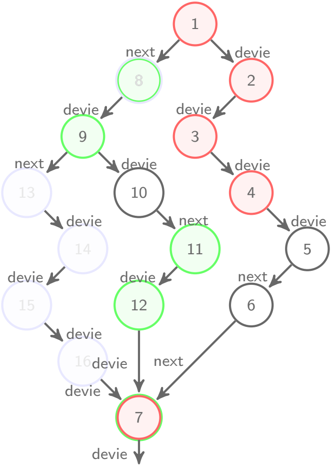
le chemin $< 1, 2, 3, 4, 5, 6, 7, \ldots, POST >$ est correct

le chemin $< 1, 8, 9, 10, 11, 12, 7, \ldots, POST >$ est correct

- À l'étape $k = 5$, notre algorithme a identifié deux déviations de correction minimales de taille égale à 5 :
  1. $D_1 = \{1, 2, 3, 4, 7\}$, le nœud "7" est marqué par la valeur 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, elle a été autorisée, car la valeur de la marque du nœud "7" est égale à la cardinalité de D_2.
- À l'étape $k = 6$, l'algorithme a suspendu la déviation suivante $D_3 = \{8, 13, 14, 15, 16, 7\}$, car la cardinalité de D_3 est supérieure strictement à la valeur de l'étiquette du nœud "7"

Notre approche

LocFaults : Illustration du déroulement du marquage des noeuds

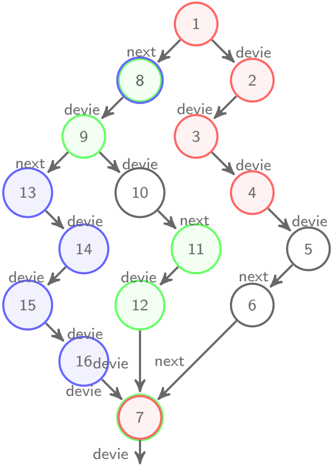


le chemin $< 1, 2, 3, 4, 5, 6, 7, \ldots, POST >$ est correct
le chemin $< 1, 8, 9, 10, 11, 12, 7, \ldots, POST >$ est correct

- À l'étape $k = 5$, notre algorithme a identifié deux déviations de correction minimales de taille égale à 5 :
  1. $D_1 = \{1, 2, 3, 4, 7\}$, le nœud "7" est marqué par la valeur 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, elle a été autorisée, car la valeur de la marque du nœud "7" est égale à la cardinalité de D_2.
- À l'étape $k = 6$, l'algorithme a suspendu la déviation suivante $D_3 = \{8, 13, 14, 15, 16, 7\}$, car la cardinalité de D_3 est supérieure strictement à la valeur de l'étiquette du nœud "7"

Notre approche

LocFaults : Illustration du déroulement du marquage des noeuds

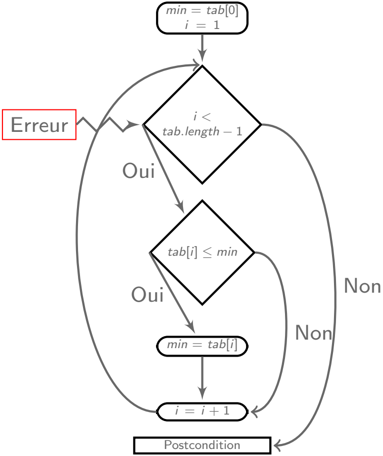


le chemin $< 1, 2, 3, 4, 5, 6, 7, \ldots, POST >$ est correct
le chemin $< 1, 8, 9, 10, 11, 12, 7, \ldots, POST >$ est correct

- À l'étape $k = 5$, notre algorithme a identifié deux déviations de correction minimales de taille égale à 5 :
  1. $D_1 = \{1, 2, 3, 4, 7\}$, le nœud "7" est marqué par la valeur 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, elle a été autorisée, car la valeur de la marque du nœud "7" est égale à la cardinalité de D_2.
- À l'étape $k = 6$, l'algorithme a suspendu la déviation suivante $D_3 = \{8, 13, 14, 15, 16, 7\}$, car la cardinalité de D_3 est supérieure strictement à la valeur de l'étiquette du nœud "7"

Notre approche

LocFaults : Illustration sur le programme avec boucle Minimum



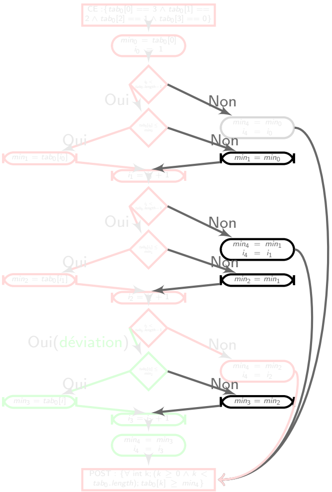
Notre approche

LocFaults : Illustration sur le programme avec boucle Minimum

- $CE = \{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$
- Le chemin initial fautif (illustration en rouge)
- La déviation pour laquelle la postcondition est satisfaite (illustration en vert)

Chemins et MCSs générés par LocFaults pour le programme Minimum :

PATH	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, min_3 = min_2, i_3 = i_2, POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]$	$\{i_0 = 1\}$, $\{i_1 = i_0 + 1\}$, $\{i_2 = i_1 + 1\}$



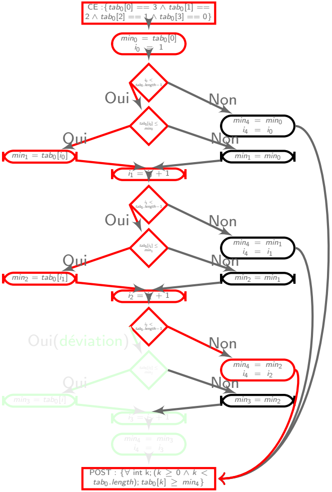
Notre approche

LocFaults : Illustration sur le programme avec boucle Minimum

- $CE = \{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$
- Le chemin initial fautif (illustration en rouge)
- La déviation pour laquelle la postcondition est satisfaite (illustration en vert)

Chemins et MCSs générés par LocFaults pour le programme Minimum :

PATH	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0]$, $min_0 = tab_0[0]$, $i_0 = 1$, $min_1 = tab_0[i_0]$, $i_1 = i_0 + 1$, $min_2 = tab_0[i_1]$, $i_2 = i_1 + 1$, $min_3 = min_2$, $i_3 = i_2$, $POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0]$, $min_0 = tab_0[0]$, $i_0 = 1$, $min_1 = tab_0[i_0]$, $i_1 = i_0 + 1$, $min_2 = tab_0[i_1]$, $i_2 = i_1 + 1$, $[\neg(i_2 \leq tab_0.length - 1)]$	$\{i_0 = 1\}$, $\{i_1 = i_0 + 1\}$, $\{i_2 = i_1 + 1\}$



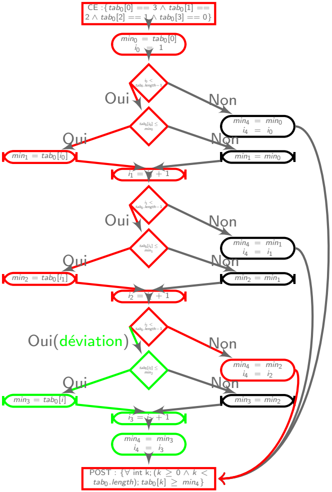
Notre approche

LocFaults : Illustration sur le programme avec boucle Minimum

- $CE = \{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$
- Le chemin initial fautif (illustration en rouge)
- La déviation pour laquelle la postcondition est satisfaite (illustration en vert)

Chemins et MCSs générés par LocFaults pour le programme Minimum :

PATH	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, min_3 = min_2, i_3 = i_2, POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]\}$	$\{i_0 = 1\}, \{i_1 = i_0 + 1\}, \{i_2 = i_1 + 1\}$



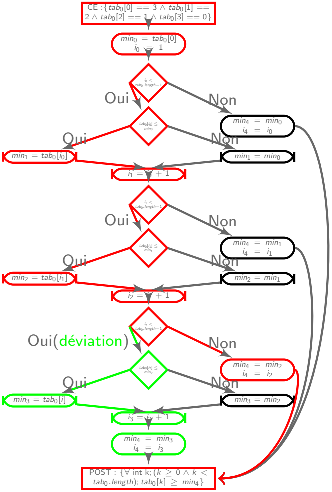
Notre approche

LocFaults : Illustration sur le programme avec boucle Minimum

- $CE = \{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$
- Le chemin initial fautif (illustration en rouge)
- La déviation pour laquelle la postcondition est satisfaite (illustration en vert)

Chemins et MCSs générés par `LocFaults` pour le programme `Minimum` :

PATH	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, min_3 = min_2, i_3 = i_2, POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]$	$\{i_0 = 1\}$, $\{i_1 = i_0 + 1\}$, $\{i_2 = i_1 + 1\}$



Évaluation expérimentale

Outils utilisés et protocole expérimental

- **LocFaults** : notre implémentation
  → Les solveurs CP OPTIMIZER et CPLEX d'IBM
  → L'outil CPBPV pour générer le CFG et CE
  → Benchmarks : les programmes Java
  → `LocFaults` se limitera à dévier au plus 3 conditions (approche incomplète)
- **BugAssist** : l'outil de localisation d'erreurs implémentant l'approche `BugAssist`
  → Le solveur MaxSAT MSUnCore2
  → L'outil CBMC pour générer le contre-exemple et la trace erronée correspondante
  → Benchmarks : les programmes ANSI-C

Évaluation expérimentale

Outils utilisés et protocole expérimental

- **LocFaults** : notre implémentation
  → Les solveurs CP OPTIMIZER et CPLEX d'IBM
  → L'outil CPBPV pour générer le CFG et CE
  → Benchmarks : les programmes Java
  → `LocFaults` se limitera à dévier au plus 3 conditions (approche incomplète)
- **BugAssist** : l'outil de localisation d'erreurs implémentant l'approche `BugAssist`
  → Le solveur MaxSAT MSUnCore2
  → L'outil CBMC pour générer le contre-exemple et la trace erronée correspondante
  → Benchmarks : les programmes ANSI-C

Évaluation expérimentale
Les programmes construits

- **Programmes académiques**
  → Programmes sans boucles et sans calculs non linéaires (minimax, caractéristiques d'un triangle)
  → Programmes sans boucles et avec calculs non linéaires (calculs sur triangle)
  → Programmes avec boucles (tri, calculs simples)
- **Un benchmark réaliste TCAS** (Traffic Collision Avoidance System)
  → 1608 cas de tests , sauf les cas de débordement du tableau *PositiveRAAltThresh*
  → V1 . . . V41

Évaluation expérimentale

Résultats sur des programmes sans boucles et calculs non linéaires

Programme	Contre-exemple	LocFaults = 0	LocFaults = 1	LocFaults = 2	LocFaults = 3	BugAssist
AbsMinusKO3	$\{i=0, j=1\}$	$\{20\}$	$\{\underline{16}\},\{14\},\{12\}$	/	/	$\{16, 20\}$
MinmaxKO	$\{in_1=2, in_2=1, in_3=3\}$	$\{10\},\{19\}$	$\{\underline{18}\},\{10\}$	/	/	$\{14, 19, 30\}$
MidKO	$\{a=2, b=1, c=3\}$	$\{19\}$	/	/	$\{\underline{14}, \underline{23}, \underline{26}\}$	$\{14, 19, 30\}$
Maxmin6varKO4	$\{a=1, b=-3, c=-4, d=-2, e=-1, f=-2\}$	$\{116\}$	/	/	$\{\underline{12}, \underline{15}, \underline{19}\}$	$\{12, 166\}$
TritypeKO	$\{i=2, j=3, k=2\}$	$\{54\}$	$\{\underline{26}\}$ $\{\underline{48}\},\{30\},\{25\}$	$\{\underline{29}, \underline{32}\}$ $\{\underline{53}, \underline{57}\},\{30\},\{25\}$	/	$\{26, 27, 32, 33, 36, 48, 57, 68\}$
TritypeKO4	$\{i=2, j=3, k=3\}$	$\{46\}$	$\{\underline{45}\},\{33\},\{25\}$	$\{\underline{26}, \underline{32}\}$	$\{\underline{32}, \underline{35}, \underline{49}\}$ $\{\underline{32}, \underline{35}, \underline{53}\}$ $\{\underline{32}, \underline{35}, \underline{57}\}$	$\{26, 27, 29, 30, 32, 33, 35, 45, 49, 68\}$
TritypeKO6	$\{i=2, j=3, k=3\}$	$\{40\}$	$\{\underline{26}\}$ $\{\underline{29}\}$	$\{\underline{35}, \underline{49}\},\{25\}$ $\{\underline{35}, \underline{53}\},\{25\}$ $\{\underline{35}, \underline{57}\},\{25\}$	/	$\{26, 27, 29, 30, 32, 33, 35, 49, 68\}$
TriPerimetreKO	$\{i=2, j=1, k=2\}$	$\{58\}$	$\{\underline{31}\}$ $\{\underline{37}\},\{32\},\{27\}$	/	/	$\{28, 29, 31, 32, 35, 37, 65, 72\}$
TriPerimetreKO6	$\{i=2, j=2, k=3\}$	$\{50\}$	$\{\underline{34}\}$ $\{\underline{37}\},\{35\},\{27\},\{29\}$	$\{\underline{49}, \underline{53}\},\{35\},\{27\},\{29\}$	/	$\{37, 72, 29, 32, 35, 34, 31, 49, 53\}$
TriPerimetreKO4	$\{i=2, j=3, k=3\}$	$\{50\}$	$\{\underline{34}\}$ $\{\underline{37}\},\{35\},\{27\}$ $\{\underline{49}\},\{35\},\{27\}$	/	/	$\{37, 35, 72, 50, 49, 34, 28, 29, 32, 61, 65, 31\}$

LocFaults fournit une localisation plus informative et plus explicative

Évaluation expérimentale

Résultats sur des programmes sans boucles et calculs non linéaires

Programme	Contre-exemple	LocFaults				BugAssist
		= 0	= 1	= 2	= 3	
AbsMinusKO3	$\{i = 0, j = 1\}$	{20}	{16},{14},{12}	/	/	{16, 20}
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	{10},{19}	{18},{10}	/	/	{14, 19, 30}
MidKO	$\{a = 2, b = 1, c = 3\}$	{19}	/	/	{14, 23, 26}	{14, 19, 30}
Maxmin6varKO4	$\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$	{116}	/	/	{12, 15, 19}	{12, 166}
TritypeKO	$\{i = 2, j = 3, k = 2\}$	{54}	{48},{26} {30},{25}	{53, 57},{29, 32} {30},{25}	/	{26, 27, 32 33, 36, 48 57, 68}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	{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}
TritypeKO6	$\{i = 2, j = 3, k = 3\}$	{40}	{26} {29}	{35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	{58}	{37},{31} {32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}
TriPerimetreKO6	$\{i = 2, j = 2, k = 3\}$	{50}	{34} {37},{35},{27},{29}	{49, 53},{35},{27},{29}	/	{37, 72, 29, 32, 35, 34, 31, 49, 53}
TriPerimetreKO4	$\{i = 2, j = 3, k = 3\}$	{50}	{34} {37},{35},{27} {49},{35},{27}	/	/	{37, 35, 72, 50, 49, 34, 28, 29, 32, 61, 65, 31}

LocFaults fournit une localisation plus informative et plus explicative

Évaluation expérimentale

Résultats (Temps de calcul)

Programmes	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO3	0.693	0.021	0.042	0.037	0.038	0.02	0.03
MinmaxKO	0.675	0.063	0.071	0.068	0.08	0.02	0.06
Maxmin6varKO4	0.785	0.029	0.03	0.034	0.05	0.07	1.05
TritypeKO	0.722	0.023	0.067	0.114	0.157	0.02	0.42
TritypeKO4	0.722	0.023	0.063	0.073	0.099	0.02	0.30
TritypeKO6	0.724	0.022	0.032	0.132	0.146	0.02	0.30
TriPerimetreKO	0.726	0.025	0.059	0.063	0.074	0.03	0.85
TriPerimetreKO4	0.727	0.025	0.117	0.124	0.102	0.04	1.12
TriPerimetreKO6	0.74	0.024	0.096	0.156	0.178	0.03	0.80

Les temps de LocFaults sont proches des temps de BugAssist

Évaluation expérimentale

Résultats (Temps de calcul)

Programmes	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO3	0.693	0.021	0.042	0.037	0.038	0.02	0.03
MinmaxKO	0.675	0.063	0.071	0.068	0.08	0.02	0.06
Maxmin6varKO4	0.785	0.029	0.03	0.034	0.05	0.07	1.05
TritypeKO	0.722	0.023	0.067	0.114	0.157	0.02	0.42
TritypeKO4	0.722	0.023	0.063	0.073	0.099	0.02	0.30
TritypeKO6	0.724	0.022	0.032	0.132	0.146	0.02	0.30
TriPerimetreKO	0.726	0.025	0.059	0.063	0.074	0.03	0.85
TriPerimetreKO4	0.727	0.025	0.117	0.124	0.102	0.04	1.12
TriPerimetreKO6	0.74	0.024	0.096	0.156	0.178	0.03	0.80

Les temps de `LocFaults` sont proches des temps de `BugAssist`

Évaluation expérimentale

Résultats (Temps de calcul pour les programmes non linéaires)

Programme	LocFaults					BugAssist	
	P	L				P	L
		$=0$	≤ 1	≤ 2	≤ 3		
TriMultPerimetreKO	0.749	0.056	0.133	0.14	0.148	0.05	3.50
TriMultPerimetreKO4	0.767	0.055	0.154	0.164	0.164	0.06	2.97
TriMultPerimetreKO5	0.756	0.056	0.113	0.166	0.177	0.05	4.09
TriMultPerimetreKO6	0.764	0.049	0.114	0.179	0.173	0.05	2.89
HeronKO	0.797	0.119	0.188	0.186	0.207	0.06	7.93
HeronKO4	0.771	0.049	0.163	0.183	0.169	0.07	4.63
HeronKO5	0.755	0.05	0.119	0.186	0.178	0.07	5.45
HeronKO6	0.746	0.048	0.115	0.176	0.179	0.07	5.03

LocFaults est plus rapide que BugAssist pour ces benchmarks

Évaluation expérimentale

Résultats (Temps de calcul pour les programmes non linéaires)

Programme	LocFaults					BugAssist	
	P	L				P	L
		$=0$	≤ 1	≤ 2	≤ 3		
TriMultPerimetreKO	0.749	0.056	0.133	0.14	0.148	0.05	3.50
TriMultPerimetreKO4	0.767	0.055	0.154	0.164	0.164	0.06	2.97
TriMultPerimetreKO5	0.756	0.056	0.113	0.166	0.177	0.05	4.09
TriMultPerimetreKO6	0.764	0.049	0.114	0.179	0.173	0.05	2.89
HeronKO	0.797	0.119	0.188	0.186	0.207	0.06	7.93
HeronKO4	0.771	0.049	0.163	0.183	0.169	0.07	4.63
HeronKO5	0.755	0.05	0.119	0.186	0.178	0.07	5.45
HeronKO6	0.746	0.048	0.115	0.176	0.179	0.07	5.03

`LocFaults` est plus rapide que `BugAssist` pour ces benchmarks

Évaluation expérimentale

Résultats (Nombre d'erreurs localisées pour TCAS)

Prog	Nb_E	Nb_CE	LF			BA
			≤ 1	≤ 2	≤ 3	
V1	1	131	131	131	131	131
V2	2	67	67	67	67	67
V3	1	23	23	23	23	13
V4	1	20	4	4	4	20
V5	1	10	9	9	9	10
V6	1	12	11	11	11	12
V7	1	36	36	36	36	36
V8	1	1	1	1	1	1
V9	1	7	7	7	7	7
V10	2	14	12	12	12	14
V11	2	14	12	12	12	14
V12	1	70	45	45	45	48
V13	1	4	4	4	4	4
V14	1	50	50	50	50	50
V16	1	70	70	70	70	70
V17	1	35	35	35	35	35
V18	1	29	28	28	28	29
V19	1	19	18	18	18	19
V20	1	18	18	18	18	18
V21	1	16	16	16	16	16
V22	1	11	11	11	11	11
V23	1	41	41	41	41	41
V24	1	7	7	7	7	7
V25	1	3	2	2	2	3
V26	1	11	7	7	7	11
V27	1	10	9	9	9	10
V28	1	75	74	74	74	58
V29	1	18	17	17	17	14
V30	1	57	57	57	57	57
V34	1	77	77	77	77	77
V35	1	75	74	74	74	58
V36	1	122	120	120	120	122
V37	1	94	21	22	22	94
V39	1	3	2	2	2	3
V40	2	122	0	72	72	122
V41	1	20	16	16	16	20

LocFaults se compare très favorablement à BugAssist

Évaluation expérimentale

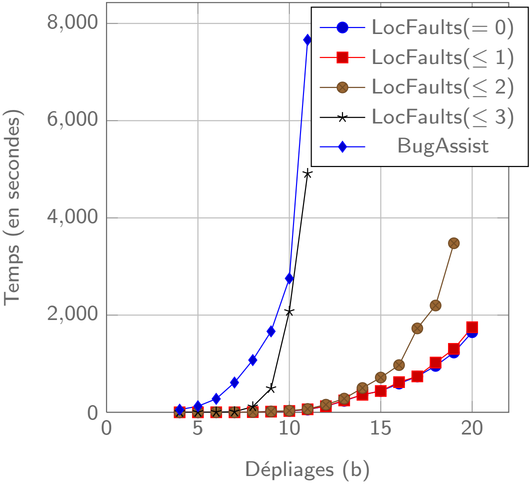
Résultats (Nombre d'erreurs localisées pour TCAS)

Prog	Nb_E	Nb_CE	LF			BA
			≤ 1	≤ 2	≤ 3	
V1	1	131	131	131	131	131
V2	2	67	67	67	67	67
V3	1	23	23	23	23	13
V4	1	20	4	4	4	20
V5	1	10	9	9	9	10
V6	1	12	11	11	11	12
V7	1	36	36	36	36	36
V8	1	1	1	1	1	1
V9	1	7	7	7	7	7
V10	2	14	12	12	12	14
V11	2	14	12	12	12	14
V12	1	70	45	45	45	48
V13	1	4	4	4	4	4
V14	1	50	50	50	50	50
V16	1	70	70	70	70	70
V17	1	35	35	35	35	35
V18	1	29	28	28	28	29
V19	1	19	18	18	18	19
V20	1	18	18	18	18	18
V21	1	16	16	16	16	16
V22	1	11	11	11	11	11
V23	1	41	41	41	41	41
V24	1	7	7	7	7	7
V25	1	3	2	2	2	3
V26	1	11	7	7	7	11
V27	1	10	9	9	9	10
V28	1	75	74	74	74	58
V29	1	18	17	17	17	14
V30	1	57	57	57	57	57
V34	1	77	77	77	77	77
V35	1	75	74	74	74	58
V36	1	122	120	120	120	122
V37	1	94	21	22	22	94
V39	1	3	2	2	2	3
V40	2	122	0	72	72	122
V41	1	20	16	16	16	20

LocFaults se compare très favorablement à BugAssist

Évaluation expérimentale

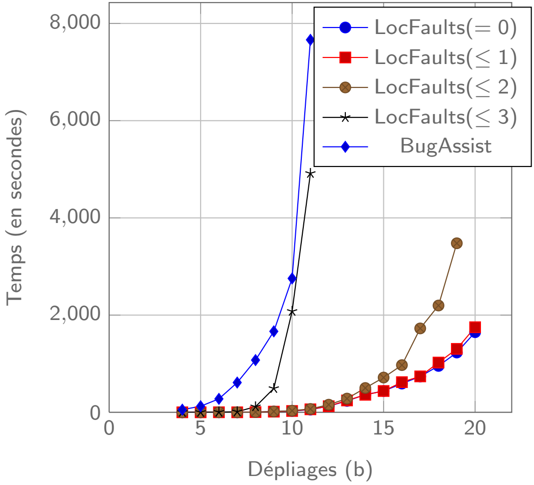
Résultats sur des programmes avec boucles : Benchmark BubbleSort



- La durée d'exécution de LocFaults et de BugAssist pour ce benchmark croît exponentiellement avec le nombre de dépliages
- Les temps de BugAssist sont toujours les plus grands
- Les différentes versions de LocFaults restent utilisables jusqu'à un certain dépliage
- Le nombre de dépliage au-delà de lequel la croissance des temps de BugAssist devient rédhibitoire est inférieur à celui de LocFaults
  → celui de LocFaults avec au plus 3 conditions déviées est inférieur à celui de LocFaults avec au plus 2 conditions déviées
  → qui est inférieur lui aussi à celui de LocFaults avec au plus 1 conditions déviées
  → les temps de LocFaults avec au plus 1 et 0 condition déviée sont presque les mêmes

Évaluation expérimentale

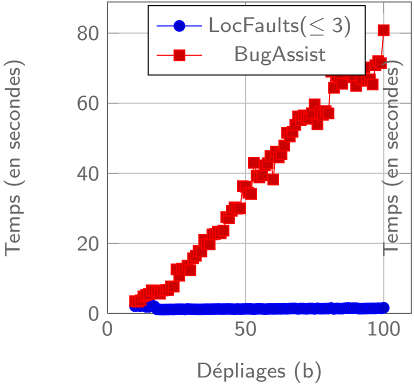
Résultats sur des programmes avec boucles : Benchmark BubbleSort

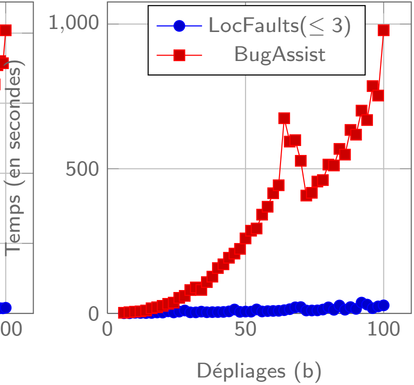


- La durée d'exécution de LocFaults et de BugAssist pour ce benchmark croît exponentiellement avec le nombre de dépliages
- Les temps de BugAssist sont toujours les plus grands
- Les différentes versions de LocFaults restent utilisables jusqu'à un certain dépliage
- Le nombre de dépliage au-delà de lequel la croissance des temps de BugAssist devient rédhibitoire est inférieur à celui de LocFaults
  → celui de LocFaults avec au plus 3 conditions déviées est inférieur à celui de LocFaults avec au plus 2 conditions déviées
  → qui est inférieur lui aussi à celui de LocFaults avec au plus 1 conditions déviées
  → les temps de LocFaults avec au plus 1 et 0 condition déviée sont presque les mêmes

Évaluation expérimentale

Résultats sur des programmes avec boucles : Sum et SquareRoot

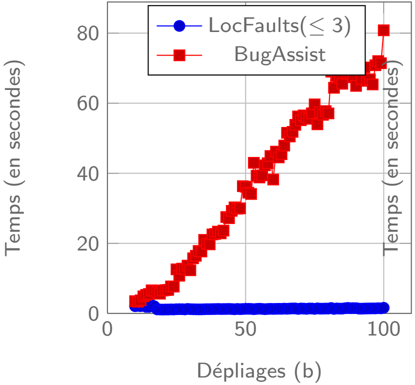


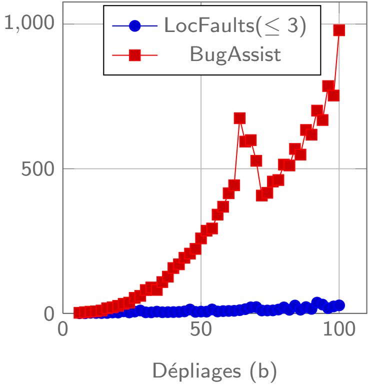


- Le temps d'exécution de BugAssist croît rapidement
- Les temps de LocFaults sont presque constants
- Les temps de LocFaults avec au plus 0, 1 et 2 conditions déviées sont proches de ceux de LocFaults avec au plus 3 conditions déviées

Évaluation expérimentale

Résultats sur des programmes avec boucles : Sum et SquareRoot





- Le temps d'exécution de `BugAssist` croît rapidement
- Les temps de `LocFaults` sont presque constants
- Les temps de `LocFaults` avec au plus 0, 1 et 2 conditions déviées sont proches de ceux de `LocFaults` avec au plus 3 conditions déviées

État de l'art

Localisation d'erreurs en test

Exemples : Tarantula, Pinpoint, Delta Debugging, WHITHER, SOBER, AMPLE

Tarantula :

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

+ Approches simples
− Besoin de **beaucoup** de cas de tests
Des approches qui requièrent l'existence d'un oracle
→ Décider si le résultat de **dizaines de milliers** de test est **juste**

Notre cadre est moins exigeant
→ Bounded Model Checking

État de l'art

Localisation d'erreurs en test

Exemples : Tarantula, Pinpoint, Delta Debugging, WHITHER, SOBER, AMPLE

Tarantula :

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

+ Approches simples

− Besoin de **beaucoup** de cas de tests

Des approches qui requièrent l'existence d'un oracle

→ Décider si le résultat de **dizaines de milliers** de test est **juste**

Notre cadre est moins exigeant

→ Bounded Model Checking

État de l'art

Localisation d'erreurs en test

Exemples : Tarantula, Pinpoint, Delta Debugging, WHITHER, SOBER, AMPLE

Tarantula :

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

+ Approches simples
− Besoin de **beaucoup** de cas de tests
  Des approches qui requièrent l'existence d'un oracle
  → Décider si le résultat de **dizaines de milliers** de test est **juste**

Notre cadre est moins exigeant
→ Bounded Model Checking

État de l'art

Localisation d'erreurs en Model Checking

Exemples : Explain, SNIPER, BugAssist

BugAssist :

- Une méthode de BMC, comme la notre
- **Différences majeures :**

  → Elle transforme la totalité du programme en une formule SAT

  → Elle est basée sur l'utilisation des solveurs MaxSAT

+ Approche globale

− Elle transforme la totalité des instructions du programme dans une formule booléenne

État de l'art

Localisation d'erreurs en Model Checking

Exemples : Explain, SNIPER, BugAssist

BugAssist :

- Une méthode de BMC, comme la notre
- **Différences majeures :**

  → Elle transforme la totalité du programme en une formule SAT

  → Elle est basée sur l'utilisation des solveurs MaxSAT

\+ Approche globale

− Elle transforme la totalité des instructions du programme dans une formule booléenne

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour calculer un seul MCS
  → BLS, FastDiag, Approche de Joao Markes-Silva

- BLS (Basic Linear Search) est un algorithme intuitif et itératif
- FastDiag se base sur le principe de diviser pour régner
- Les techniques proposées par Joao Markes-Silva peuvent être intégrées dans les algorithmes calculant les MCSs pour améliorer leurs performances

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour calculer un seul MCS

  → BLS, FastDiag, Approche de Joao Markes-Silva

- BLS (Basic Linear Search) est un algorithme intuitif et itératif
- FastDiag se base sur le principe de diviser pour régner
- Les techniques proposées par Joao Markes-Silva peuvent être intégrées dans les algorithmes calculant les MCSs pour améliorer leurs performances

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour isoler un seul MUS

  → Deletion Filter, Méthode Additive, Méthode Additive + Deletion, QuickXplain

- Les quatre algorithmes se basent sur le test de la faisabilité d'un CSP
- La différence entre eux réside dans le nombre de tests de faisabilité effectués :

La méthode		Le meilleur cas	Le pire cas
Deletion Filter		n	n
Additive Method		$\sum_{i=1}^{k} i + 1$	$\sum_{i=0}^{k-1}(n-i) + 1$
Additive/Deletion Method		$k + (k-1)$	$n + (n-1)$
QUICKXPLAIN	$split(n) = n/2$ $split(n) = n-1$ $split(n) = 1$	$log(n/k) + 2k$ $2k$ k	$2k * log(n/k) + 2k$ $2n$ $n + k$

La cardinalité de l'ensemble de contraintes en entrée est n, et la cardinalité de l'ensemble retourné est k

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour isoler un seul MUS

  → Deletion Filter, Méthode Additive, Méthode Additive + Deletion, QuickXplain

- Les quatre algorithmes se basent sur le test de la faisabilité d'un CSP
- La différence entre eux réside dans le nombre de tests de faisabilité effectués :

La méthode		**Le meilleur cas**	**Le pire cas**
Deletion Filter		n	n
Additive Method		$\sum_{i=1}^{k} i+1$	$\sum_{i=0}^{k-1}(n-i)+1$
Additive/Deletion Method		$k+(k-1)$	$n+(n-1)$
QUICKXPLAIN	$split(n)=n/2$	$log(n/k)+2k$	$2k*log(n/k)+2k$
	$split(n)=n-1$	$2k$	$2n$
	$split(n)=1$	k	$n+k$

La cardinalité de l'ensemble de contraintes en entrée est n, et la cardinalité de l'ensemble retourné est k

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour trouver plusieurs MCSs
  → Approche de M.H.Liffiton

- L'énumération des MCSs trouve de nombreuse applications, exemples : MaxSAT, l'énumération des MUSs.
- La plupart des approches existantes utilisent ou bien un solveur MaxSAT ou bien ou des appels itératifs à un solveur normal (SAT ou de contraintes)
- CAMUS qui se base sur MaxSAT est parmi les approches les plus efficaces
  1. Elle résout une série de problèmes de MaxSAT successifs
  2. Chacun avec la restriction qui exclut les MCSs trouvés précédemment
  3. Critère d'arrêt : Il ne reste plus d'ensembles satisfaisables

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour trouver plusieurs MCSs

  → Approche de M.H.Liffiton

- L'énumération des MCSs trouve de nombreuse applications, exemples : MaxSAT, l'énumération des MUSs.
- La plupart des approches existantes utilisent ou bien un solveur MaxSAT ou bien ou des appels itératifs à un solveur normal (SAT ou de contraintes)
- CAMUS qui se base sur MaxSAT est parmi les approches les plus efficaces
  1. Elle résout une série de problèmes de MaxSAT successifs
  2. Chacun avec la restriction qui exclut les MCSs trouvés précédemment
  3. Critère d'arrêt : Il ne reste plus d'ensembles satisfaisables

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour trouver de multiples MUSs
  → DAA, CAMUS, MARCO

- DAA et CAMUS exploitent la relation entre MCSs et MUSs
- Les différences majeures entre DAA et CAMUS :
  1. CAMUS calcule tous les MCSs, puis il calcule tous les MUSs
  2. DAA calcule les deux tout au long de son exécution
  3. CAMUS souffre du fait que le nombre des MCSs peut être exponentiel
     →Impact négatif pour faire des progrès sur le calcul des MUSs
  4. DAA est mieux, mais il ne pourrait pas trouver un MUS de taille k s'il n'a pas trouvé au moins k MCSs
- CAMUS peut énumérer tous les MUSs plus rapidement que MARCO, tandis que MARCO est plus approprié pour le calcul de certains MUSs rapidement

État de l'art

Calcul MCS et MUS dans un système de contraintes inconsistant

- Algorithmes pour trouver de multiples MUSs

  → DAA, CAMUS, MARCO

- DAA et CAMUS exploitent la relation entre MCSs et MUSs
- Les différences majeures entre DAA et CAMUS :
  1. CAMUS calcule tous les MCSs, puis il calcule tous les MUSs
  2. DAA calcule les deux tout au long de son exécution
  3. CAMUS souffre du fait que le nombre des MCSs peut être exponentiel

     →Impact négatif pour faire des progrès sur le calcul des MUSs
  4. DAA est mieux, mais il ne pourrait pas trouver un MUS de taille k s'il n'a pas trouvé au moins k MCSs
- CAMUS peut énumérer tous les MUSs plus rapidement que MARCO, tandis que MARCO est plus approprié pour le calcul de certains MUSs rapidement

Conclusions & perspectives

Conclusions

- 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
  - Exploration DFS + Calcul des MCSs bornés
    → Pour empêcher l'explosion combinatoire
- Basée sur l'utilisation des solveurs de contraintes
- `LocFaults` fournit des résultats plus précis et plus pertinents par rapport à `BugAssist`
  → Les temps de `LocFaults` sont meilleurs pour les programmes avec calculs numériques
- `LocFaults` localise fréquemment les erreurs pour les programmes TCAS

Conclusions & perspectives

Perspectives

- Expérimenter plus de programmes réels
- Calcul des MUSs
- Traiter les instructions avec calcul sur les flottants
  - → Utiliser des **solveurs spécialisés** sur les flottants
- Mesurer le degré de suspicion de chaque instruction
- Méthode hybride, statistique et BMC, pour la localisation d'erreurs

Merci pour votre attention !

Questions ?