Une approche CSP pour l'aide à la localisation d'erreurs

Mohammed Bekkouche, Hélène Collavizza, Michel Rueher

▶ **To cite this version:**

Mohammed Bekkouche, Hélène Collavizza, Michel Rueher. Une approche CSP pour l'aide à la localisation d'erreurs. 2014. hal-00957255v1

HAL Id: hal-00957255

https://hal.science/hal-00957255v1

Submitted on 9 Mar 2014 (v1), last revised 25 Apr 2014 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Une approche CSP pour l'aide à la localisation d'erreurs*

Mohammed Bekkouche Hélène Collavizza Michel Rueher

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
{helen,bekkouche,rueher}@unice.fr

Résumé

Nous proposons dans cet article une nouvelle approche basée sur la CP pour l'aide à la localisation des erreurs dans un programme. Nous supposons qu'un contre-exemple a été trouvé, c'est à dire que l'on dispose d'une instanciation des variables d'entrée qui viole la post-condition. Pour aider à localiser les erreurs, nous générons un système de contraintes pour les chemins du CFG (Graphe de Flot de Contrôle) où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces chemins des ensembles minima de correction (ou MCS - Minimal Correction Set) de taille bornée. Le retrait d'un de ces ensembles de contraintes produit un MSS (Maximal Satisfiable Subset) qui ne viole plus la post condition. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah [19] afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques. Nous présentons les résultats des premières expérimentations qui sont encourageants.

Abstract

We introduce in this paper a new CP-based approach to support errors location in a program for which a counter-example is available, i.e. an instantiation of the input variables that violates the post-condition. To provide helpful information for error location, we generate a constraint system for the paths of the CFG (Control Flow Graph) for which at most k conditional statements may be erroneous. Then, we calculate Minimal Correction Sets (MCS) of bounded size for each of these paths. The removal of one of these sets of constraints yields a maximal satisfiable subset, in other words, a maximal subset of constraints satisfying the post condition. We extend the algorithm proposed by Liffiton and Sakallah [19] to handle programs with numerical statements more efficiently. We present preliminary experimental results that are quite encouraging.

*Ce travail a débuté au NII (National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430) où Michel Rueher a été invité plusieurs fois en tant que Professeur associé depuis 2011. Les idées générales et la démarche ont été définies lors des réunions de travail avec les professeurs Hiroshi HOSOBE et Shin NAKAJIMA.

1 Introduction

L'aide à la localisation d'erreur à partir de contre-exemples ou de traces d'exécution est une question cruciale lors de la mise au point de logiciels critiques. En effet, quand un programme P contient des erreurs, un model-checker fournit un contre-exemple ou une trace d'exécution qui est souvent longue et difficile à comprendre, et de ce fait d'un intérêt très limité pour le programmeur qui doit débugger son programme. La localisation des portions de code qui contiennent des erreurs est donc souvent un processus difficile et couteux, même pour des programmeurs expérimentés. C'est pourquoi nous proposons dans cet article une nouvelle approche basée sur la CP pour l'aide à la localisation des erreurs dans un programme pour lequel un contre-exemple à été trouvé ; c'est à dire pour lequel on dispose d'une instanciation des variables d'entrée qui viole la post-condition. Pour aider à localiser les erreurs, nous générons un système de contraintes pour les chemins du CFG (Graphe de Flot de Contrôle) où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces chemins des ensembles minima de correction (ou MCS - Minimal Correction Set) de taille bornée. Le retrait d'un de ces ensembles des contraintes initiales produit un MSS (Maximal Satisfiable Subset) qui ne viole plus la post condition. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques. Nous présentons les résultats des premières expérimentations qui sont encourageants.

La prochaine section est consacrée à un positionnement de notre approche par rapport aux principales méthodes qui ont été proposées pour ce résoudre ce problème. La section suivante est dédiée à la description de notre approche et des algorithmes utilisés. Puis, nous présentons les résultats des premières expérimentations avant de conclure.

2 État de l'art

Dans cette section, nous positionnons notre approche par rapport aux principales méthodes existantes. Nous parlerons d'abord des méthodes utilisées pour l'aide à la localisation des erreurs dans le cadre du test et de la vérification de programmes. Comme l'approche que nous proposons consiste essentiellement à rechercher des sous-ensembles de contraintes spécifiques dans un système de contraintes inconsistant, nous parlerons aussi dans un second temps des algorithmes qui ont été utilisés en recherche opérationnelle et en programmation par contraintes pour aider l'utilisateur à debugger un système de contraintes inconsistant.

2.1 Méthodes d'aide à la localisation des erreurs utilisées en test et en vérification de programmes

Différentes approches ont été proposées pour aider le programmeur dans l'aide à la localisation d'erreur dans la communauté test et vérification.

Bal et al[1] utilisent plusieurs appels à un Model Checker et comparent les contre-exemples obtenus avec une trace d'exécution correcte. Les transitions qui ne figurent pas dans la trace correcte sont signalées comme une possible cause de l'erreur. Ils ont implanté leur algorithme dans le contexte de SLAM, un model checker qui vérifie les propriétés de sécurité temporelles de programmes C.

Plus récemment, des approches basées sur la dérivation de traces correctes ont été introduites dans un système nommé `Explain`[13, 12] et qui fonctionne en trois étapes :

1. Appel de `CBMC` [1] pour trouver une exécution qui viole la post-condition ;
2. Utilisation d'un solveur pseudo-booléen pour rechercher l'exécution correcte la plus proche ;
3. Calcul de la différence entre les traces.

`Explain` produit ensuite une formule propositionnelle S associée à `P` mais dont les affectations ne violent pas la spécification. Enfin `Explain` étend S avec des contraintes représentant un problème d'optimisation : trouver une affectation satisfaisante qui soit aussi proche que possible du contre-exemple ; la proximité étant mesurée par une distance sur les exécutions de `P`.

Une approche similaire à `Explain` a été introduite dans [22] mais elle est basée sur le test plutôt que sur la vérification de modèle : les auteurs utilisent des séries de tests correctes et des séries erronées. Ils utilisent aussi des métriques de distance pour sélectionner un test correct à partir d'un ensemble donné de tests. Cette approche suppose qu'un oracle soit disponible.

Dans [10, 11], les auteurs partent aussi de la trace d'un contre-exemple, mais ils utilisent la spécification pour dériver un programme correct pour les mêmes données d'entrée. Chaque instruction identifiée est un candidat potentiel de faute et elle peut être utilisée pour corriger les erreurs. Cette approche garantit que les erreurs sont effectivement parmi les instructions identifiées (en supposant que l'erreur est dans le modèle erroné considéré). En d'autres termes, leur approche identifie un sur-ensemble des instructions erronées. Pour réduire le nombre d'erreurs potentielles, le processus est redémarré pour différents contre-exemples et les auteurs calculent l'intersection des ensembles d'instructions suspectes. Cependant, cette approche souffre de deux problèmes majeurs :

- Elle permet de modifier n'importe quelle expression, l'espace de recherche peut ainsi être très grand ;
- Elle peut renvoyer beaucoup de faux diagnostics totalement absurdes car toute modification d'expression est possible (par exemple, changer la dernière affectation d'une fonction pour renvoyer le résultat attendu).

Pour remédier à ces inconvénients, Zhang et al [25] proposent de modifier uniquement les prédicats de flux de contrôle. L'intuition de cette approche est qu'à travers un switch des résultats d'un prédicat et la modification du flot de contrôle, l'état de programme peut non seulement être modifié à peu de frais, mais qu'en plus, il est souvent possible d'arriver à un état succès. Liu et al [20] généralisent cette approche en permettant la modification de plusieurs prédicats. Ils proposent également une étude théorique d'un algorithme de débogage pour les erreurs *RHS*, c'est à dire les erreurs dans les prédicats de contrôle et la partie droite des affectations.

Dans [2], les auteurs abordent le problème de l'analyse de la trace d'un contre-exemple et de l'identification de l'erreur dans le cadre des systèmes de vérification formelle du hardware. Ils utilisent pour cela la notion de causalité introduite par Halpern et Pearl pour définir formellement une série de causes de la violation de la spécification par un contre-exemple.

Récemment, Manu Jose et Rupak Majumdar [14, 15] ont abordé ce problème différemment : ils ont introduit un nouvel algorithme qui utilise un solveur MAX-SAT pour calculer le nombre maximum des clauses d'une formule booléenne qui peut être satisfaite par une affectation. Leur algorithme fonctionne en trois étapes :

1. Comme dans l'exécution symbolique, ils encodent une trace d'un programme par une formule booléenne F qui est satisfiable si et seulement si la trace est satisfiable ;
2. ils construisent une formule fausse F' en imposant que la post-condition soit vraie (la formule F' est insatisfiable car la trace correspond à un contre-exemple qui viole la post-condition) ;
3. Ils utilisent `MAXSAT` pour calculer le nombre maximum de clauses pouvant être satisfaites dans F' et affichent le complément de cet ensemble comme une cause potentielle des erreurs. En d'autres termes, ils calculent le complément d'un MSS (Maximal Satisfiable Subset).

Manu Jose et Rupak Majumdar [14, 15] ont implanté leur algorithme dans un outil appelé `BugAssist` qui utilise `CBMC`.

1. `http://www.cprover.org/cbmc/`

Si-Mohamed Lamraoui et Shin Nakajima [17] ont aussi développé récemment un outil nommé `SNIPER` qui calcule les MSS d'une formule $\psi = EI \wedge TF \wedge AS$ ou EI encode les valeurs d'entrée erronées, TF est une formule qui représente tous les chemins du programme, et AS correspond à l'assertion qui est violée. Les MCS sont obtenus en prenant le complément des MSS calculés. L'implémentation est basée sur la représentation intermédiaire `LLVM` et le solveur SMT `Yices`. L'implémentation actuelle est toutefois beaucoup plus lente que `BugAssist`.

L'approche que nous proposons ici est inspirée des travaux de Manu Jose et Rupak Majumdar. Les principales différences sont :

1. Nous ne transformons pas tout le programme en un système de contraintes mais nous utilisons le graphe de flot de contrôle pour collecter les contraintes qui correspondent aux instructions du chemin d'un contre exemple.
2. Nous n'utilisons pas des algorithmes basés sur `MAXSAT` mais des algorithmes plus généraux qui permettent plus facilement de traiter des contraintes numériques.

2.2 Méthodes pour debugger un système de contraintes inconsistant

En recherche opérationnelle et en programmation par contraintes, différents algorithmes ont été proposés pour aider l'utilisateur à debugger un système de contraintes inconsistant. Lorsqu'on recherche des informations utiles pour la localisation des erreurs sur les systèmes de contraintes numériques, on peut s'intéresser à deux types d'informations :

1. Combien de contraintes dans un ensemble de contraintes insatisfiables peuvent être satisfaites ?
2. Où se situe le problème dans le système de contraintes ?

Avant de présenter rapidement les algorithmes [2] qui cherchent à répondre à ces questions, nous allons définir plus formellement les notion de MUS, MSS et MCS à l'aide des définitions introduites dans [19].
Soit C un ensemble de contraintes :

$$M \subseteq C \text{ est un MUS} \Leftrightarrow M \text{ est UNSAT}$$
$$\text{et } \forall c \in M : M \setminus \{c\} \text{ est SAT.}$$

La notion de MSS (Maximal Satisfiable Subset) est une généralisation de MaxSAT / MaxCSP où l'on considère la maximalité au lieu de la cardinalité maximale :

$$M \subseteq C \text{ est un MSS} \Leftrightarrow M \text{ est SAT}$$
$$\text{et } \forall c \in C \setminus M : M \cup \{c\} \text{ est UNSAT.}$$

Cette définition est très proche de celle des IIS (Irreducible Inconsistent Subsystem) utilisés en recherche opérationnelle [4, 5, 6].
Les MCS (Minimal Correction Set) sont des compléments des MSS (le retrait d'un MCS à C produit un MSS car on "corrige" l'infaisabilité) :

$$M \subseteq C \text{ est un MCS} \Leftrightarrow C \setminus M \text{ est SAT}$$
$$\text{et } \forall c \in M : (C \setminus M) \cup \{c\} \text{ est UNSAT.}$$

Il existe donc une dualité entre l'ensemble des MUS et des MCS [3, 19] : informellement, l'ensemble des MCS est équivalent aux ensembles couvrants irréductibles [3] des MUS ; et l'ensemble des MUS est équivalent aux ensembles couvrants irréductibles des MCS. Soit un ensemble de contraintes C :

1. Un sous-ensemble M de C est un MCS ssi M est un ensemble couvrant minimal des MUS de C ;
2. Un sous-ensemble M de C est un MUS ssi M est un ensemble couvrant minimal des MCS de C ;

Au niveau intuitif, il est aisé de comprendre qu'un MCS doit au moins retirer une contrainte de chaque MUS. Et comme un MUS peut être rendu satisfiable en retirant n'importe laquelle de ses contraintes, chaque MCS doit au moins contenir une contrainte de chaque MUS. Cette dualité est aussi intéressante pour notre problématique car elle montre que les réponses aux deux questions posées ci-dessus sont étroitement liées.

Différents algorithmes ont été proposés pour le calcul des IIS/MUS et MCS. Parmi les premiers travaux, ont peut mentionner les algorithmes `Deletion Filter`, `Additive Method`, `Additive Deletion Method`, `Elastic Filter` qui ont été développés dans la communauté de recherche opérationnelle [4, 24, 5, 6]. Les trois premiers algorithmes sont des algorithmes itératifs alors que le quatrième utilise systématiquement des variables d'écart pour identifier dans la première phase du Simplexe les contraintes susceptibles de figurer dans un IIS.
Junker [16] a proposé un algorithme générique basé sur une stratégie "Divide-and-Conquer" pour calculer efficacement les IIS/MUS lorsque la taille des sous-ensembles conflictuels est beaucoup plus petite que celle de l'ensemble total des contraintes.
L'algorithme de Liffiton et Sakallah [19] qui calcule d'abord l'ensemble des MCS par ordre de taille croissante, puis l'ensemble des MUS est basé sur la propriété mentionnée ci-dessus. Cet algorithme, que nous avons utilisé dans notre implémentation est décrit dans la section suivante.

Différentes améliorations [9, 18, 21] de ces algorithmes ont été proposées ces dernières années mais elles sont assez étroitement liées à SAT et donc plus difficilement transposables dans un contexte où nous avons de nombreuses contraintes numériques.

3 Notre approche

Dans cette section nous allons d'abord présenter le cadre général de notre approche, à savoir celui du "Bounded Model Checking" (BMC) basé sur la programmation par contraintes, puis nous allons décrire la méthode proposée et les algorithmes utilisés pour calculer des MCS de cardinalité bornée.

2. Pour une présentation plus détaillée voir `http://users.polytech.unice.fr/~rueher/Publis/Talk_NII_2013-11-06.pdf`

3. On rappelle que H est un ensemble couvrant de Σ si $H \subseteq D$ et $\forall S \in \Sigma : H \cup S \neq \emptyset$. H est irréductible (ou minimal) si aucun élément ne peut être retiré de H sans que celui-ci ne perde sa propriété d'ensemble couvrant.

3.1 Les principes : BMC et MCS

Notre approche se place dans le cadre du "Bounded model Checking" (BMC) par programmation par contraintes [7, 8]. En BMC, les programmes sont dépliés en utilisant une borne b, c'est à dire que les boucles sont remplacées par des imbrications de conditionnelles de profondeur au plus b. Il s'agit ensuite de détecter des non-conformités par rapport à une spécification. Étant donné un triplet de Hoare $\{PRE, PROG_b, POST\}$, où PRE est la pré-condition, $PROG_b$ est le programme déplié b fois et $POST$ est la post-condition, le programme est *non conforme* si la formule $\Phi = PRE \wedge PROG_b \wedge \neg POST$ est satisfiable. Dans ce cas, une instanciation des variables de Φ est un *contre-exemple*, et un cas de non conformité, puisqu'il satisfait à la fois la pré-condition et le programme, mais ne satisfait pas la post-condition.

`CPBPV` [7] est un outil de BMC basé sur la programmation par contraintes. $CPBPV$ transforme PRE et $POST$ en contraintes, et transforme $PROG_b$ en un CFG (Control Flow Graph) dans lequel les conditions et les affectations sont traduites en contraintes [4]. `CPBPV` construit le CSP de la formule Φ *à la volée*, par un parcours en profondeur du graphe. À l'état initial, le CSP contient les contraintes de PRE et $\neg POST$, puis les contraintes d'un chemin sont ajoutées au fur et à mesure de l'exploration du graphe. Quand le dernier noeud d'un chemin est atteint, la faisabilité du CSP est testée. S'il est consistant, alors on a trouvé un contre-exemple, sinon, un retour arrière est effectué pour explorer une autre branche du CFG. Si tous les chemins ont été explorés sans trouver de contre-exemple, alors le programme est conforme à sa spécification (sous réserve de l'hypothèse de dépliage des boucles).

Les travaux présentés dans cet article cherchent à *localiser* l'erreur détectée par la phase de BMC. Plus précisemment, soit CE une instanciation des variables qui satisfait le CSP contenant les contraintes de PRE et $\neg POST$, et les contraintes d'un chemin incorrect de $PROG_b$ noté $PATH$. Alors le CSP $C = CE \cup PRE \cup PATH \cup POST$ est *inconsistant*, puisque CE est un contre-exemple et ne satisfait donc pas la post-condition. Un *ensemble minima de correction* (ou MCS - Minimal Correction Set) de C est un ensemble de contraintes qu'il faut nécessairement enlever pour que C devienne consistant. Un tel ensemble fournit donc une *localisation de l'erreur* sur le chemin du contre-exemple. Comme l'erreur peut se trouver dans une affectation sur le chemin du contre-exemple, mais peut aussi provenir d'un mauvais branchement, notre approche (nommée `LocFaults`) s'intéresse également aux MCS des systèmes de contraintes obtenus en déviant un branchement par rapport au comportement induit par le contre-exemple. Plus précisément, $LocFaults$ effectue un parcours en profondeur d'abord du CFG de $PROG_b$, en propageant le contre-exemple et en déviant au plus une condition :

- Quand aucune condition n'est déviée, alors `LocFaults` a parcouru le chemin du contre-exemple en collectant les contraintes de ce chemin. Il calcule ensuite les MCS sur cet ensemble de contraintes,
- Quand une condition c_d est déviée, et que le chemin résultant ne viole plus la post-condition, deux cas sont possibles :
  (i) la condition elle-même est cause de l'erreur,
  (ii) une erreur dans une affectation a provoqué une mauvaise évaluation de c_d, faisant prendre le mauvais branchement.
  Dans le cas (ii), le CSP $PRE \cup PATH_{c_d} \cup \{c_d\}$, où $PATH_{c_d}$ est l'ensemble des contraintes du chemin partant du premier noeud du CFG jusqu'à la condition c_d, est satisfiable. Par conséquent, le CSP $PRE \cup PATH_{c_d} \cup \{\neg c_d\}$ est *insatisfiable*. `LocFaults` calcule donc également les MCS de ce CSP, afin de détecter les instructions suspectées d'avoir induit le mauvais branchement pour c_d.

Cette approche peut naturellement être généralisée au cas où k mauvais branchements ont été effectués. Dans la suite de l'article, on se restreindra toutefois au cas où une seule condition est susceptible d'être fausse.

3.2 Description de l'algorithme

L'algorithme `LocFaults` (cf. Algorithm 1) prend en entrée un programme déplié non conforme vis-à-vis de sa spécification, un contre-exemple, et une borne maximum de la taille des ensembles de correction. Il dévie au plus une condition par rapport au contre-exemple fourni, et renvoie une liste de corrections possibles.

`LocFaults` commence par construire le CFG du programme (*ligne 3*) puis appelle la fonction `DFS` qui gère deux ensembles de contraintes :

- C : l'ensemble des contraintes du chemin (chemin du contre-exemple ou chemin jusqu'à la condition déviée),
- P : l'ensemble des contraintes dites de propagation, c'est à dire les contraintes de la forme $variable = constante$ qui sont obtenues en propageant le contre exemple sur les affectations du chemin.

L'ensemble P est utilisé pour propager les informations et vérifier si une condition est satisfaite, l'ensemble C est utilisé pour calculer les MCS. Ces deux ensembles sont collectés à la volée lors du parcours en profondeur. Les paramètres de la fonction `DFS` sont les ensembles C et P décrits ci-dessus, n le noeud courant du CFG, c_d la contrainte déviée ($\perp$ si aucune condition n'est déviée), MCS la liste des corrections en cours de construction, et MCS_b la borne de la taille des MCS. Nous notons $n.left$ (resp. $n.right$) la branche *if* (resp. *else*) d'un noeud conditionnel, et $n.next$ le noeud qui suit un bloc d'affectation ; *cstr* est la fonction qui traduit une condition ou affectation en contraintes.

Le parcous commence avec C vide et P contenant les contraintes du contre-exemple. Il part de la racine de l'arbre ($CFG.root$) qui contient la pré-condition, et se déroule comme suit :

- Quand le dernier noeud est atteint (i.e. noeud de la post-condition, *lignes 10 à 27*), alors si aucune

4. Pour éviter les problèmes de re-définitions multiples des variables, la forme DSA est utilisée

Algorithm 1: LocFaults

```
1  Fonction LocFaults(PROG_b,CE,MCS_b)
   Entrées: PROG_b : un programme déplié b fois non
            conforme vis-à-vis de sa spécification, CE : un
            contre-exemple de PROG_b,MCS_b : la borne du
            cardinal des MCS
   Sorties: une liste de corrections possibles
2  début
3      CFG ← CFG_build(PROG_b)
4      MCS = []
5      DFS(∅,CE,CFG.root,⊥,MCS,MCS_b)
6      retourner MCS
7  fin

8  DFS(C,P,n,c_d,MCS,MCS_b)
   Entrées: C : contraintes collectées, P : contraintes de
            propagation, n : noeud du CFG, c_d : la
            condition déviée (⊥ si vide), MCS : ensemble
            des MCS calculés (initialisé à ∅),MCS_b : la
            borne du cardinal des MCS
9  début
10     si n est la postcondition alors
11         si c_d=⊥ alors
12             % aucune condition déviée
13             % calcul des MCS sur le chemin du
               contre-exemple
14             C ← C ∪ {cstr(POST)}
15             MCS.add(MCS(C, MCS_b))
16         sinon
17             % on a dévié une condition
18             si P ∪ {cstr(POST)} est faisable alors
19                 % le chemin dévié est correct
20                 % la condition déviée est une correction
21                 MCS.add({c_d})
22                 % calcul des MCS sur le chemin qui mène
                   à la condition déviée
23                 C ← C ∪ {¬c_d}
24                 MCS.add({c_d} ∪ MCS(C, MCS_b))
25             fin
26         fin
27     fin
28     sinon si n est un noeud conditionnel alors
29         si P ∪ {cstr(n.cond)} est faisable alors
30             % la propagation du contre-exemple satisfait
                 la condition, exploration de la branche If
31             DFS(C,P, n.left,c_d,MCS,MCS_b)
32             si c_d = ⊥ alors
33                 % On dévie la condition et on explore la
                   branche Else
34                 DFS(C,P, n.right,n.cond,MCS,MCS_b)
35             fin
36         fin
37         sinon
38             DFS(C,P, n.right,c_d,MCS,MCS_b) % On
               explore la branche Else
39             si c_d=⊥ alors
40                 % On dévie la condition et on explore la
                   branche If
41                 DFS(C,P, n.left,n.cond,MCS,MCS_b)
42             fin
43         fin
44     fin
45     sinon si (n est un bloc d'affectations) alors
46         % on propage le contre-exemple
47         pour chaque affectation ass ∈ n.assigns faire
48             P.add(propagate(ass,CE))
49         fin
50         si (c_d=⊥) alors
51             % On n'a pas encore pris de déviation
52             % on ajoute les contraintes qui mènent à la
                 prochaine déviation
53             pour chaque affectation ass ∈ n.assigns
               faire
54                 C ← C ∪ {cstr(ass)}
55             fin
56         fin
57         % On continue l'exploration sur le noeud suivant
58         DFS(C,P, n.next,c_d,MCS,MCS_b)
59     fin
60 fin
```

Algorithm 2: MCS

```
1  Fonction MCS(C,MCS_b)
   Entrées: C : Ensemble de contraintes infaisable,
            MCS_b : Entier
   Sorties: MCS : Liste de MCS de C de
            cardinalité inférieure à MCS_b
2  début
3      C' ← AddYVars(C)
4      MCS ← ∅
5      k ← 1
6      tant que SAT(C') ∧ k<=MCS_b faire
7          C'_k ← C' ∧
           AtMost({¬y_1, ¬y_2, ..., ¬y_n},k)
8          tant que SAT(C'_k) faire
9              MCS.add(newMCS).
10             C'_k ← C'_k ∧
               BlockingClause(newMCS)
11             C' ← C' ∧
               BlockingClause(newMCS).
12         fin
13         k ← k + 1.
14     fin
15     retourner MCS
16 fin
```

condition n'a été déviée (*lignes 11 à 15*), on ajoute la post-condition à C et on cherche les MCS. Si une condition a été déviée, et que cela corrige le programme, (*lignes 18 à 25*) alors on ajoute la condition (*ligne 21*), et on cherche aussi les MCS sur le chemin menant à cette condition (*lignes 23 et 24*),

- Quand le noeud est un noeud conditionnel (*lignes 28 à 44*), alors on utilise P pour savoir si la condition est satisfaite. Si oui on prend la branche correspondante. Si aucune condition n'a encore été déviée, alors on dévie cette condition pour essayer l'autre branche.
- Quand le noeud est un bloc d'affectations (*lignes 45 à 59*), on propage le contre-exemple sur ces affectations et on ajoute les contraintes correspondantes dans P. Si aucune contrainte n'a été déviée, c'est à dire que l'on est sur un chemin qui précéde une future déviation, on ajoute les contraintes de l'affectation dans C.

L'algorithme `LocFaults` appelle l'algorithme `MCS` (cf. Algorithm 2) qui est une transcription directe de l'algorithme proposé par Liffiton et Sakallah [19]. Cet algorithme associe à chaque contrainte un sélecteur de variable y_i qui peut prendre la valeur 0 ou 1 ; la contrainte `AtMost` permet donc de retenir au plus k contraintes du système de contraintes initial dans le MCS. La procédure `BlockingClause(newMCS)` appelée à la ligne 10 (resp. ligne 11) permet d'exclure les sur-ensembles de taille k (resp. de taille supérieure à k).

Lors de l'implémentation de cet algorithme nous avons utilisé `IBM ILOG CPLEX` [5] qui permet à la fois une implémentation aisée de la fonction `AtMost` et la résolution de systèmes de contraintes numériques. Il faut toutefois noter que cette résolution n'est correcte que sur les entiers et que la prise en compte des

5. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

nombres flottants nécessite l'utilisation d'un solveur spécialisé pour le traitement des flottants.

4 Evaluation expérimentale

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de `LocFaults` et de `BugAssist` [14, 15] sur un ensemble de programmes. Comme `LocFaults` est basé sur `CPBPV`[7] qui travaille sur des programmes Java et que`BugAssist` travaille sur des programmes C, nous avons construit pour chacun des programmes :

- une version en Java annotée par une spécification JML ;
- une version en ANSI-C annotée par la même spécification mais en ACSL.

Les deux versions ont les mêmes numéros de ligne et les mêmes instructions. La précondition est un contre-exemple du programme, et la postcondition correspond au résultat de la version correcte du programme pour les données du contre-exemple. Comme dit précédemment, nous avons considéré qu'au plus une condition pouvait être fausse sur un chemin. Par ailleurs, nous n'avons pas cherché de MCS de cardinalité supérieure à 3. Les expérimentations ont été effectuées avec un processeur Intel Core i7-3720QM 2.60 GHz avec 8 GO de RAM.

Nous avons d'abord utilisé un ensemble de programmes académiques de petite taille (entre 15 et 100 lignes). A savoir :

- **AbsMinus**. Ce programme prend en entrée deux entiers i et j et renvoie la valeur absolue de $i - j$.
- **Minmax**. Ce programme prend en entrée trois entiers : $in1$, $in2$ et $in3$, et permet d'affecter la plus petite valeur à la variable *least* et la plus grande valeur à la variable *most*.
- **Tritype**. Ce programme est un programme classique qui a été utilisé très souvent en test et vérification de programmes. Il prend en entrée trois entiers (les côtés d'un triangle) et retourne 3 si les entrées correspondent à un triangle équilatéral, 2 si elles correspondent à un triangle isocèle, 1 si elles correspondent à un autre type de triangle, 4 si elles ne correspondent pas à un triangle valide.
- **TriPerimetre**. Ce programme a exactement la même structure de contrôle que tritype. La différence est que TriPerimetre renvoie la somme des côtés du triangle si les entrées correspondent à un triangle valide, et -1 dans le cas inverse.

Pour chacun de ces programmes, nous avons considéré différentes versions érronées.

Nous avons aussi évalué notre approche sur les programmes TCAS (Traffic Collision Avoidance System) de la suite de test Siemens[23]. Il s'agit là aussi d'un benchmark bien connu qui correspond à un système d'alerte de trafic et d'évitement de collisions aériennes. Il y a 41 versions erronées et 1608 cas de tests. Nous avons utilisé toutes les versions erronées sauf celles dont l'indice $AltLayerValue$ déborde du tableau $PositiveRAAltThresh$ car les débordements de tableau ne sont pas traités dans `CPBPV`. Plus précisément, voici les versions **TcasKO...TcasKO41**. Les erreurs dans ces programmes sont provoquées dans des endroits différents. 1608 cas de tests sont proposés, chacun correspondant à contre-exemple. Pour chacun de ces cas de test T_j, on construit un programme $TcasV_iT_j$ qui prend comme entrée le contre-exemple, et dont postcondition correspond à la sortie correcte attendue.

Le code source de l'ensemble des programmes est disponible à l'adresse `http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html`.

La table 1 contient les résultats pour le premier ensemble de programmes :

- Pour `LocFaults` nous affichons la liste des MCS. La première ligne correspond aux MCS identifiés sur le chemin initial. Les lignes suivantes aux MCS identifiés sur les chemins pour lesquels la postcondition est satisfaite lorsqu'une condition est déviée. Le numéro de la ligne correspondant à la condition est souligné, ceux des instructions d'affectation qu'il faut modifier pour prendre le chemin sans modifier l'expression de la condition sont en italiques.
- Pour `BugAssist` les résultats correspondent à la fusion de l'ensemble des compléments des MSS calculés, fusion qui est opérée par `BugAssist` avant l'affichage des résultats.

Sur ces benchmarks les résultats de `LocFaults` sont plus concis et plus précis que ceux de `BugAssist`.

La table 2 fournit les temps de calcul : dans les deux cas, P correspond au temps de prétraitement et L au temps de calcul des MCS. Pour `LocFaults`, le temps de pré-traitement inclut la traduction du programme Java en un arbre de syntaxe abstraite avec l'outil JDT (Eclipse Java development tools), insi que la construction du CFG dont les noeuds sont des ensembles de contraintes. C'est la traduction Java qui est la plus longue. Pour `BugAssist`, le temps de prétraitement est celui la construction de la formule SAT. On remarquera que `BugAssist` est plus rapide que `LocFaults` pour les programmes qui ne contiennent quasiment pas d'opérations arithmétiques, alors que c'est l'inverse pour les différentes versions de `TriPerimetre` qui contiennent un peu plus de calcul.

La table 3 donne les résultats pour les programmes de la suite `TCAS`. La colonne Nb_E indique pour chaque programme le nombre d'erreurs qui ont été introduites dans le programme alors que la colonne Nb_CE donne le nombre de contre-exemples. Les colonnes LF et BA indiquent le nombre de fois où `LocFaults` et `BugAssist` ont identifié l'instruction erronée. On remarquera que `LocFaults` se compare favorablement à `BugAssist` sur ce benchmark qui ne contient quasiment aucune instruction arithmétique ; comme précédemment le nombre d'instructions suspectes identifiées par `LocFaults` est dans l'ensemble nettement inférieur à celui de `BugAssist`.

Les temps de calcul de `BugAssist` et `LocFaults` sont très similaires et inférieurs à une seconde pour chacun des benchmarks de la suite `TCAS`.

Programme	Contre-exemple	Erreur	LocFaults	BugAssist
AbsMinusKO	$\{i = 0, j = 1\}$	17	$\{17\}$	$\{17\}$
AbsMinusKO2	$\{i = 0, j = 1\}$	11	$\{11\}$, $\{17\}$	$\{16, 17, 20\}$
AbsMinusKO3	$\{i = 0, j = 1\}$	14	$\{20\}$ $\{16\}$ $\{12\}$, $\{14\}$	$\{16, 20\}$
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	19	$\{10\}$, $\{19\}$ $\{18\}$ $\{10\}$	$\{18, 19, 22\}$
TritypeKO	$\{i = 2, j = 3, k = 2\}$	54	$\{54\}$ $\{48\}$ $\{25\}$, $\{30\}$ $\{26\}$	$\{26, 27, 32, 33, 36, 48, 57, 68\}$
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	$\{54\}$ $\{53\}$ $\{25\}$, $\{27\}$ $\{35\}$ $\{25\}$, $\{27\}$ $\{26\}$ $\{21\}$	$\{21, 26, 27, 29, 30,$ $32, 33, 35, 36, 53, 68\}$
TritypeKO2V2	$\{i = 1, j = 2, k = 1\}$	31	$\{50\}$ $\{49\}$ $\{25\}$, $\{31\}$ $\{36\}$ $\{25\}$, $\{31\}$ $\{29\}$ $\{26\}$ $\{21\}$	$\{21, 26, 27, 29, 31,$ $33, 34, 36, 37, 49, 68\}$
TritypeKO3	$\{i = 1, j = 2, k = 1\}$	53	$\{54\}$ $\{53\}$ $\{25\}$, $\{30\}$ $\{35\}$ $\{25\}$, $\{30\}$ $\{29\}$ $\{21\}$	$\{21, 26, 27, 29, 30, 32,$ $33, 35, 36, 48, 53, 68\}$
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	$\{46\}$ $\{45\}$ $\{25\}$, $\{33\}$	$\{26, 27, 29, 30, 32, 33,$ $35, 45, 49, 68\}$
TritypeKO5	$\{i = 2, j = 3, k = 3\}$	32, 45	$\{40\}$ $\{29\}$ $\{26\}$	$\{26, 27, 29, 30, 32, 33,$ $35, 49, 68\}$
TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	58	$\{58\}$ $\{37\}$ $\{27\}$, $\{32\}$ $\{31\}$	$\{28, 29, 31, 32, 35, 37, 65, 72\}$
TriPerimetreKOV2	$\{i = 2, j = 3, k = 2\}$	34	$\{34\}$, $\{60\}$ $\{40\}$ $\{27\}$, $\{33\}$ $\{32\}$	$\{28, 32, 33, 34, 36, 38, 40, 41,$ $52, 55, 56, 60, 64, 67, 74\}$
TriPerimetreKO2	$\{i = 1, j = 1, k = 2\}$	57	$\{58\}$ $\{57\}$ $\{27\}$, $\{29\}$ $\{37\}$ $\{27\}$, $\{29\}$ $\{28\}$ $\{22\}$	$\{22, 28, 29, 31, 32, 34, 35, 37,$ $38, 48, 49, 52, 53, 57, 58, 61, 72\}$
TriPerimetreKO3	$\{i = 1, j = 2, k = 1\}$	57	$\{58\}$ $\{57\}$ $\{27\}$, $\{32\}$ $\{37\}$ $\{27\}$, $\{32\}$ $\{31\}$ $\{22\}$	$\{22, 28, 29, 31, 32, 34, 35, 37,$ $38, 49, 52, 57, 72\}$

TABLE 1 – MCSes identifiés par LocFaults et BugAssist

Programme	LocFaults		BugAssist	
	P	L	P	L
AbsMinusKO	$0,692s$	$0,054s$	$0,02s$	$0,03s$
AbsMinusKO2	$0,694s$	$0,073s$	$0,01s$	$0,06s$
AbsMinusKO3	$0,717s$	$0,117s$	$0,02s$	$0,04s$
MinmaxKO	$0,672s$	$0,259s$	$0,01s$	$0,09s$
TritypeKO	$0,719s$	$0,126s$	$0,03s$	$0,42s$
TritypeKO2	$0,722s$	$0,259s$	$0,02s$	$1,00s$
TritypeKO2V2	$0,714s$	$0,274s$	$0,02s$	$0,82s$
TritypeKO3	$0,718s$	$0,241s$	$0,02s$	$0,71s$
TritypeKO4	$0,709s$	$0,113s$	$0,02s$	$0,33s$
TritypeKO5	$0,721s$	$0,096s$	$0,01s$	$0,36s$
TriPerimetreKO	$0,734s$	$0,128s$	$0,03s$	$0,89s$
TriPerimetreKOV2	$0,72s$	$0,154s$	$0,03s$	$1,68s$
TriPerimetreKO2	$0,726s$	$0,295s$	$0,07s$	$2,87s$
TriPerimetreKO3	$0,722s$	$0,255s$	$0,03s$	$1,58s$

TABLE 2 – Temps de calcul

5 Discussion

Nous avons présenté dans cet article une nouvelle approche pour l'aide à la localisation d'erreurs qui utilise quelques spécificités de la programmation par contraintes. Les premiers résultats sont encourageants mais doivent encore être confirmés sur des programmes plus importants et contenant plus d'opérations arithmétiques.

Les travaux futurs concernent à la fois une réflexion sur le traitement des boucles (dans un cadre de bounded-model checking) et l'évaluation de notre approche lorsqu'on suppose que plus d'une condition peut être fausse.

D'un point de vue applicatif, il sera certainement aussi intéressant de calculer des MUS car ils apportent une information complémentaire à l'utilisateur.

Remerciements :

Nous tenons à remercier Hiroshi Hosobe, Yahia Lebbah, Si-Mohamed Lamraoui et Shin Nakajima pour les échanges fructueux que nous avons eus. Nous tenons aussi à remercier Olivier Ponsini pour son aide et ses précieux conseils lors de la réalisation du prototype de notre système.

Références

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause : localizing errors in counterexample traces. In Proceedings of POPL, pages 97–105. ACM, 2003.

[2] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. In

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

TABLE 3 – Nombre d'erreurs localisés pour TCAS

Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009.

[3] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. J. Exp. Theor. Artif. Intell., 15(1) :25–46, 2003.

[4] John W. Chinneck. Localizing and diagnosing infeasibilities in networks. INFORMS Journal on Computing, 8(1) :55–62, 1996.

[5] John W. Chinneck. Fast heuristics for the maximum feasible subsystem problem. INFORMS Journal on Computing, 13(3) :210–223, 2001.

[6] John W. Chinneck. Feasibility and Infeasibility in Optimization : Algorithms and Computational Methods. Springer, 2008.

[7] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. Constraints, 15(2) :238–264, 2010.

[8] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher, and Antoine Rollet. Constraint-based bmc : a backjumping strategy. STTT, 16(1) :103–121, 2014.

[9] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM, 26(1) :53–62, 2012.

[10] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In Proceedings of CAV, volume 4144 of Lecture Notes in Computer Science, pages 358–371. Springer, 2006.

[11] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. Electr. Notes Theor. Comput. Sci., 174(4) :95–111, 2007.

[12] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. STTT, 8(3) :229–247, 2006.

[13] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In Proceedings of CAV, volume 3114 of Lecture Notes in Computer Science, pages 453–456. Springer, 2004.

[14] Manu Jose and Rupak Majumdar. Bug-assist : Assisting fault localization in ansi-c programs. In Proceedings of CAV, volume 6806 of Lecture Notes in Computer Science, pages 504–509. Springer, 2011.

[15] Manu Jose and Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. In Proceedings of PLDI, pages 437–446. ACM, 2011.

[16] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In Proceedings of AAAI, pages 167–172. AAAI Press / The MIT Press, 2004.

[17] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization in imperative programs. NII research report, Submitted For publication, 6 pages, February, 2014.

[18] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility : Finding multiple muses quickly. In Proc. of CPAIOR, volume 7874 of Lecture Notes in Computer Science, pages 160–175. Springer, 2013.

[19] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning, 40(1) :1–33, 2008.

[20] Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In Proceedings of AAAI. AAAI Press, 2010.

[21] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013.

[22] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In Proceedings of ASE, pages 30–39. IEEE Computer Society, 2003.

[23] David S. Rosenblum and Elaine J. Weyuker. Lessons learned from a regression testing case study. Empirical Software Engineering, 2(2) :188–191, 1997.

[24] Mehrdad Tamiz, Simon J. Mardle, and Dylan F. Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. Computers & OR, 23(2) :113–119, 1996.

[25] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006.