Le livrable L5.3 a été rédigé par l'I3S. Il explore une nouvelle approche basée sur la programmation par contraintes pour l'aide à la localisation des erreurs dans un programme pour lequel une instanciation des variables d'entrée qui viole une post-condition est disponible. Pour identifier un ensemble d'instruction suspectes nous calculons des ensembles minima de correction (ou MCS - Minimal Correction Set) de taille bornée. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques.

Le livrable se présente sous la forme de deux articles joints en annexe de ce document

Références

- Une approche CSP pour l'aide à la localisation d'erreurs, Mohammed Bekkouche, Hélène Collavizza, Michel Rueher, Dixièmes Journées Francophones de Programmation par Contraintes (JFPC 14), Jun 2014,Angers, France. http://jfpc-jiaf2014.univ-angers.fr/jfpc/
- LocFaults: A new flow-driven and constraint-based error localization approach. . Mohammed Bekkouche, Hélène Collavizza, Michel Rueher. ACM SAC'15, SVT track, Apr 2015, Salamanca, Spain. HAL : hal-01094227

Annexe 1:

Une approche CSP pour l'aide à la localisation d'erreurs,
Mohammed Bekkouche, Hélène Collavizza, Michel Rueher,
Dixièmes Journées Francophones de Programmation par Contraintes (JFPC 14), Jun 2014,Angers, France.
http://jfpc-jiaf2014.univ-angers.fr/jfpc/

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 pour lequel un contre-exemple est disponible, 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 [22] 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 [22] to handle programs with numerical statements more efficiently. We present preliminary experimental results that are quite encouraging.

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 coûteux, 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 a 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

*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.

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'erreurs dans la communauté test et vérification.

Le problème de la localisation des erreurs a d'abord été abordé dans le cadre du test où de nombreux systèmes ont été développés. Le plus célèbre est Tarantula [16, 15] qui utilise différentes métriques pour établir un classement des instructions suspectes détectées lors de l'exécution d'une batterie de tests. Le point critique de cette approche réside dans le fait qu'elle requiert un oracle qui permet de décider si le résultat du test est juste ou non. Nous nous plaçons ici dans un cadre moins exigeant (et plus réaliste) qui est celui du Bounded-Model Checking (BMC), c'est à dire un cadre où les seuls prérequis sont un programme, une post-condition ou une assertion qui doit être vérifiée, et éventuellement une pré-condition.

C'est aussi dans ce cadre que se placent Bal et al[1] qui 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`[14, 13] 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 [25] mais elle est basée sur le test plutôt que sur la vérification de modèles : 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 [11, 12], 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 [28] 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 [23] 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 [3], 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 [17, 18] 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. 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

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

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 [17, 18] ont implanté leur algorithme dans un outil appelé `BugAssist` qui utilise `CBMC`.

Si-Mohamed Lamraoui et Shin Nakajima [20] ont aussi développé récemment un outil nommé `SNIPER` qui calcule les MSS d'une formule $\psi = EI \wedge TF \wedge AS$ où 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 du chemin du contre exemple et des chemins dérivés de ce dernier en supposant qu'au plus k instructions conditionnelles sont susceptibles de contenir des erreurs.
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 débugger 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 [22].
Un MUS (Minimal Unsatisfiable Subsets) est un ensemble de contraintes qui est inconsistant mais qui devient consistant si une contrainte quelconque est retirée de cet ensemble. Plus formellement, 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 [5, 6, 7].
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 [4, 22] : 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 [5, 27, 6, 7]. Les trois premiers algorithmes sont des algorithmes itératifs alors que le quatrième utilise des variables d'écart pour identifier dans la première phase du Simplexe les contraintes susceptibles de figurer dans un IIS.
Junker [19] 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 [22] 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 [10, 21, 24] de ces algorithmes ont été proposées ces dernières années mais elles sont assez étroitement liées à SAT et a priori assez

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

3. Soit Σ un ensemble d'ensemble et D l'union des élements de Σ. 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.

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.

3.1 Les principes : BMC et MCS

Notre approche se place dans le cadre du "Bounded model Checking" (BMC) par programmation par contraintes [8, 9]. 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` [8] 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 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 des branchements par rapport au comportement induit par le contre-exemple. Plus précisément, l'algorithme $LocFaults$ effectue un parcours en profondeur d'abord du CFG de $PROG_b$, en propageant le contre-exemple et en déviant au plus k_{max} conditions. Trois cas peuvent être distingués :

- Aucune condition n'a été déviée : `LocFaults` a parcouru le chemin du contre-exemple en collectant les contraintes de ce chemin et il va calculer les MCS sur cet ensemble de contraintes ;
- k_{max} conditions ont été déviées sans qu'on arrive à trouver un chemin qui satisfasse la post-condition : on abandonne l'exploration de ce chemin ;
- d conditions on déja été déviées et on peut encore dévier au moins une condition, c'est à dire $k_{max} > 1$. Alors la condition courante c est déviée. Si le chemin résultant ne viole plus la post-condition, l'erreur sur le chemin initial peut avoir deux causes différentes :
  (i) les conditions déviées elles-mêmes sont cause de l'erreur,
  (ii) une erreur dans une affectation a provoqué une mauvaise évaluation de c, faisant prendre le mauvais branchement.

  Dans le cas (ii), le `CSP` $CE \cup PRE \cup PATH_c \cup \{c\}$, où $PATH_c$ est l'ensemble des contraintes du chemin courant, c'est à dire le chemin du contre-exemple dans lequel d déviations ont déjà été prises, est satisfiable. Par conséquent, le `CSP` $CE \cup PRE \cup PATH_c \cup \{\neg c\}$ 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.

  Bien entendu, nous ne calculons pas les MCS des chemins ayant le même préfixe : si la déviation d'une condition permet de satisfaire la post-condition, il est inutile de chercher à modifier des conditions supplémentaires dans la suite du chemin

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 k_{max} conditions par rapport au contre-exemple fourni, et renvoie une liste de corrections possibles.

`LocFaults` commence par construire le CFG du programme puis appelle la fonction `DFS` sur le chemin du contre-exemple (i.e. en déviant 0 condition) puis en acceptant au plus k_{max} déviations. La fonction `DFS` gère trois ensembles de contraintes :

- CSP_d : l'ensemble des contraintes des conditions

4. Pour éviter les problèmes de re-définitions multiples des variables, la forme DSA (Dynamic Single Assignment [2]) est utilisée

Algorithm 1: LocFaults

```
1  Fonction LocFaults(PROG_b,CE,k_max,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,
   – k_max : le nombre maximum de conditions à dévier,
   – MCS_b : la borne du cardinal des MCS
   Sorties: une liste de corrections possibles
2  début
3      CFG ← CFG_build(PROG_b) % construction du CFG
4      MCS = []
5      DFS_devie(CFG.root,CE,∅,∅,0,MCS,MCS_b) % calcul des MCS sur le chemin du contre-exemple
6      DFS_devie(CFG.root,CE,∅,∅,k_max,MCS,MCS_b) % calcul des MCS en prenant au plus k_max déviations
7      retourner MCS
8  fin
```

Algorithm 2: DFS_{devie}

```
1  Fonction DFS_devie(n,P,CSP_d,CSP_a,k,MCS,MCS_b)
   Entrées:
   – n : noeud du CFG,
   – P : contraintes de propagation (issues du contre-exemple et du chemin),
   – CSP_d : contraintes des conditions déviées,
   – CSP_a : contraintes des affectations,
   – k : nombre de conditions à dévier,
   – MCS : ensemble des MCS calculés,
   – MCS_b : la borne du cardinal des MCS
2  début
3      si n est la postcondition alors
4          % on est sur le chemin du CE, calcul des MCS
5          CSP_a ← CSP_a ∪ {cstr(POST)}
6          MCS.add(MCS(CSP_a, MCS_b))
7      fin
8      sinon si n est un noeud conditionnel alors
9          si P ∪ {cstr(n.cond)} est faisable alors
10             % next est le noeud où l'on doit aller, devie est la branche opposée
11             next = n.gauche
12             devie = n.droite
13         fin
14         sinon
15             next = n.droite
16             devie = n.gauche
17         fin
18         si k > 0 alors
19             % on essaie de dévier la condition courante
20             corrige = correct(devie,P)
21             si corrige alors
22                 % le chemin est corrigé, on met à jour les MCS
23                 CSP_d ← CSP_d ∪ {cstr(n.cond)}
24                 MCS.addAll(CSP_d) % ajout des conditions déviées
25                 % calcul des MCS sur le chemin qui mène à la dernière condition déviée
26                 pour chaque c dans CSP_d faire
27                     CSP_a ← CSP_a ∪ {¬c}
28                 fin
29                 MCS.add(MCS(CSP_a, MCS_b))
30             fin
31             sinon si k>1 alors
32                 % on essaie de dévier la condition courante et des conditions en dessous
33                 DFS_devie(devie,P,CSP_d ∪ {cstr(n.cond)},CSP_a,k − 1,MCS,MCS_b)
34             fin
35             % dans tous les cas, on essaie de dévier les conditions en dessous du noeud courant
36             DFS_devie(next,P,CSP_d,CSP_a,k,MCS,MCS_b)
37         fin
38         sinon
39             % k=0, on est sur le chemin du contre-exemple, on suit le chemin
40             DFS_devie(next,P,CSP_d,CSP_a,k,MCS,MCS_b)
41         fin
42     fin
43     sinon si (n est un bloc d'affectations) alors
44         pour chaque affectation ass ∈ n.assigns faire
45             P.add(propagate(ass,P))
46             CSP_a ← CSP_a ∪ {cstr(ass)}
47         fin
48         % On continue l'exploration sur le noeud suivant
49         DFS_devie(n.next,CE,P,CSP_d,CSP_a,k,MCS,MCS_b)
50     fin
51 fin
```

qui ont été déviées à partir du chemin du contre-exemple,

- CSP_a : l'ensemble des contraintes d'affectations du chemin,
- 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

Algorithm 3: correct

```
1  Fonction correct(n,P)
   Entrées:
   – n : noeud du CFG,
   – P : contraintes de propagation (issues du contre-exemple et
     du chemin),
   Sorties: true si le programme est correct sur le chemin
            induit par P
2  début
3      si n est la postcondition alors
4          si P ∪ {cstr(POST)} est faisable alors
5              retourner true
6          fin
7          sinon
8              retourner false
9          fin
10     fin
11     sinon si n est un noeud conditionnel alors
12         si P ∪ {cstr(n.cond)} est faisable alors
13             % exploration de la branche If
14             retourner correct(n.left,P)
15         fin
16         sinon
17             retourner correct(n.right,P)
18         fin
19     fin
20     sinon si (n est un bloc d'affectations) alors
21         % on propage les affectations
22         pour chaque affectation ass ∈ n.assigns faire
23             P.add(propagate(ass,P))
24         fin
25         % On continue l'exploration sur le noeud suivant
26         retourner
           correct(n.next,P,CSP_d,CSP_a,MCS,MCS_b)
27     fin
28 fin
```

Algorithm 4: 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
```

chemin.

L'ensemble P est utilisé pour propager les informations et vérifier si une condition est satisfaite, les ensembles CSP_d et CSP_a sont utilisés pour calculer les MCS. Ces trois ensembles sont collectés à la volée lors du parcours en profondeur. Les paramètres de la fonction `DFS` sont les ensembles CSP_d, CSP_a et P décrits ci-dessus, n le noeud courant du CFG, MCS la liste des corrections en cours de construction, k le nombre de déviations autorisées 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 parcours commence avec CSP_d et CSP_a vides 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), on est sur le chemin du contre-exemple. La post-condition est ajoutée à CSP_a et on cherche les MCS,
- Quand le noeud est un noeud conditionnel, alors on utilise P pour savoir si la condition est satisfaite. Si on peut encore prendre une déviation (i.e. $k > 0$), on essaie de dévier la condition courante c et on vérifie si cette déviation corrige le programme en appelant la fonction *correct*. Cette fonction propage tout simplement le contre-exemple sur le graphe à partir du noeud courant et renvoie vrai si le programme satisfait la post-condition pour ce chemin,
  - Si dévier c a corrigé le programme, alors les conditions qui ont été déviées (i.e. $CSP_d \cup c$) sont des corrections. De plus, on calcule aussi les corrections dans le chemin menant à c,
  - Si dévier c n'a pas corrigé le programme, si on peut encore dévier des conditions (i.e. $k \geq 1$) alors on dévie c et on essaie de dévier $k - 1$ conditions en dessous de c.

  Dans les deux cas (dévier c a corrigé ou non le programme), on essaie aussi de dévier des conditions en dessous de c, sans dévier c.
- Quand le noeud est un bloc d'affectations, on propage le contre-exemple sur ces affectations et on ajoute les contraintes correspondantes dans P et dans CSP_a.

L'algorithme `LocFaults` appelle l'algorithme `MCS` (cf. Algorithm 4) qui est une transcription directe de l'algorithme proposé par Liffiton et Sakallah [22]. 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 nombres flottants nécessite l'utilisation d'un solveur spécialisé pour le traitement des flottants.

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

Programme	Contre-exemple	Erreurs	LocFaults				BugAssist
			$= 0$	≤ 1	≤ 2	≤ 3	
AbsMinusKO	$\{i = 0, j = 1\}$	17	{17}	{17}	{17}	{17}	{17}
AbsMinusKO2	$\{i = 0, j = 1\}$	11	{11},{17}	{11},{17}	{11},{17}	{11},{17}	{17, 20, 16}
AbsMinusKO3	$\{i = 0, j = 1\}$	14	{20}	{16},{14},{12}, {20}	{16},{14},{12}, {20}	{16},{14},{12}, {20}	{16, 20}
MinmaxKO	$\{in_1 = 2, in_2 = 1,$ $in_3 = 3\}$	19	{10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18, 19, 22}
MidKO	$\{a = 2, b = 1, c = 3\}$	19	{19}	{19}	{19}	{14, 23, 26} {19}	{14, 19, 30}
Maxmin6varKO	$\{a = 1, b = -4, c = -3,$ $d = -1, e = 0, f = -4\}$	27	{28}	{15} {27} {28}	{15} {27} {28}	{15} {27} {28}	{15, 12, 27, 31, 166}
Maxmin6varKO2	$\{a = 1, b = -3, c = 0,$ $d = -2, e = -1, f = -2\}$	12	{65}	{12} {65}	{12} {65}	{12} {65}	{12, 64, 166}
Maxmin6varKO3	$\{a = 1, b = -3, c = 0,$ $d = -2, e = -1, f = -2\}$	12,15	{65}	{65}	{12, 15} {65}	{12, 15} {65}	{12, 15, 64, 166}
Maxmin6varKO4	$\{a = 1, b = -3, c = -4,$ $d = -2, e = -1, f = -2\}$	12,15, 19	{116}	{116}	{116}	{12, 15, 19} {116}	{12, 166}
TritypeKO	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48} {30},{25} {54}	{26} {29, 32} {48},{30},{25} {53, 57},{25},{30} {54}	{26} {29, 32} {29, 35, 57},{25} {32, 44, 57},{33}, {25},{30} {48},{30},{25} {53, 57},{25},{30} {54}	{26, 27, 32, 33, 36, 48, 57, 68}
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27}, {25} {32, 44},{33},{25}, {27} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27}, {25} {32, 44},{33},{25}, {27} {35},{27},{25} {53},{25},{27} {54}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 53, 68}
TritypeKO2V2	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25} {50}	{21} {26} {29} {33, 45},{34},{31}, {25} {36},{31},{25} {49},{31},{25} {50}	{21} {26} {29} {33, 45},{34},{31}, {25} {36},{31},{25} {49},{31},{25} {50}	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
TritypeKO3	$\{i = 1, j = 2, k = 1\}$	53	{54}	{21} {29} {35},{30},{25} {53},{30},{25} {54}	{21} {26, 57},{30},{25}, {27} {29} {32, 44},{33},{30}, {25} {35},{30},{25} {53},{30},{25} {54}	{21} {26, 57},{30},{25}, {27} {29} {32, 44},{33},{30}, {25} {35},{30},{25} {53},{30},{25} {54}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 48, 53, 68}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45},{33},{25} {46}	{26, 32} {29, 32} {45},{33},{25} {46}	{26, 32} {29, 32} {32, 35, 49},{25} {32, 35, 53},{25} {32, 35, 57},{25} {45},{33},{25} {46}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TritypeKO5	$\{i = 2, j = 3, k = 3\}$	32,45	{40}	{26} {29} {40}	{26} {29} {32, 45},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26} {29} {32, 45},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TritypeKO6	$\{i = 2, j = 3, k = 3\}$	32,33	{40}	{26} {29} {40}	{26} {29} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26} {29} {32, 45, 49},{33},{25} {32, 45, 53},{33},{25} {32, 45, 57},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26, 27, 29, 30, 32, 33 35, 49, 68}
TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{28, 29, 31, 32, 35, 37, 65, 72}
TriPerimetreKOV2	$\{i = 2, j = 3, k = 2\}$	34	{60},{34}	{32} {40},{33},{27} {60}, {34}	{32} {40},{33},{27} {60}, {34}	{32} {40},{33},{27} {60}, {34}	{28, 32, 33, 34, 36, 38, 40, 41, 52, 55, 56, 60, 64, 67, 74}

TABLE 1 – MCS identifiés par LocFaults pour des programmes sans boucles

4 Evaluation expérimentale

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de `LocFaults` et de `BugAssist` [17, 18] sur un ensemble de programmes. Comme `LocFaults` est basé sur `CPBPV`[8] 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. Nous avons considéré qu'au plus trois conditions pouvaient être fausses 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*.

Programme	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO	$0,487s$	$0,044s$	$0,073s$	$0,074s$	$0,062s$	$0,02s$	$0,03s$
AbsMinusKO2	$0,484s$	$0,085s$	$0,065s$	$0,085s$	$0,078s$	$0,01s$	$0,06s$
AbsMinusKO3	$0,479s$	$0,076s$	$0,113s$	$0,357s$	$0,336s$	$0,02s$	$0,04s$
MinmaxKO	$0,528s$	$0,243s$	$0,318s$	$0,965s$	$1,016s$	$0,01s$	$0,09s$
MidKO	$0,524s$	$0,065s$	$0,078s$	$0,052s$	$0,329s$	$0,02s$	$0,08s$
Maxmin6varKO	$0,528s$	$0,082s$	$0,132s$	$0,16s$	$0,149s$	$0,06s$	$1,07s$
Maxmin6varKO2	$0,536s$	$0,064s$	$0,072s$	$0,097s$	$0,126s$	$0,06s$	$0,66s$
Maxmin6varKO3	$0,545s$	$0,066s$	$0,061s$	$0,29s$	$0,307s$	$0,04s$	$1,19s$
Maxmin6varKO4	$0,538s$	$0,06s$	$0,07s$	$0,075s$	$0,56s$	$0,04s$	$0,78s$
TritypeKO	$0,493s$	$0,022s$	$0,097s$	$0,276s$	$2,139s$	$0,03s$	$0,35s$
TritypeKO2	$0,51s$	$0,023s$	$0,25s$	$2,083$	$3,864s$	$0,02s$	$0,69s$
TritypeKO2V2	$0,514s$	$0,034s$	$0,28s$	$1,178s$	$1,31s$	$0,02s$	$0,77s$
TritypeKO3	$0,493s$	$0,022s$	$0,26s$	$1,928s$	$4,535s$	$0,02$	$0,48s$
TritypeKO4	$0,497s$	$0,023s$	$0,095s$	$0,295$	$5,127s$	$0,02s$	$0,21s$
TritypeKO5	$0,492s$	$0,021s$	$0,099s$	$0,787s$	$0,8s$	$0,01s$	$0,25s$
TritypeKO6	$0,492s$	$0,025s$	$0,078s$	$0,283s$	$1,841s$	$0,03s$	$0,24s$
TriPerimetreKO	$0,518s$	$0,047s$	$0,126s$	$1,096s$	$2,389s$	$0,03s$	$0,64s$
TriPerimetreKOV2	$0,503s$	$0,043s$	$0,271s$	$0,639s$	$1,958s$	$0,03s$	$1,20s$

TABLE 2 – Temps de calcul

- **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[26]. 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`. A savoir, 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.
- 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), ainsi 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. Globalement, les performances de `LocFaults` et `BugAssist` sont similaires bien que le processus d'évaluation de nos systèmes de contraintes soit loin d'être optimisé.

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 respectivement le nombre de contre-exemples pour lesquels `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 `Loc-`

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

`Faults` 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`.

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.

Au niveau des résultats obtenus `LocFaults` est plus précis que `BugAssist` lorsque les erreurs sont sur le chemin du contre exemple ou dans une des conditions du chemin du contre-exemple. Ceci provient du fait que `BugAssist` et `LocFaults` ne calculent pas exactement la même chose :

`BugAssist` calcule les compléments des différents sous-ensembles obtenus par `MaxSat,` c'est à dire des sous-ensembles de clauses satisfiables de cardinalité maximale. Certaines "erreurs " du programme ne vont pas être identifiées par `BugAssist` car les contraintes correspondantes ne figurent pas dans le complément d'un sous ensemble de clauses satisfiables de cardinalité maximale.

`LocFaults` calcule des MCS, c'est à dire le complément d'un sous-ensemble de clauses maximal, c'est à dire auquel on ne peut pas ajouter d'autre clause sans le rendre inconsistant, mais qui n'est pas nécessairement de cardinalité maximale.

`BugAssist` identifie des instructions suspectes dans l'ensemble du programme alors que `LocFaults` recherche les instructions suspectes sur un seul chemin.

Les travaux futurs concernent à la fois une réflexion sur le traitement des boucles (dans un cadre de bounded-model checking) et l'optimisation de la résolution pour les contraintes numériques. On utilisera aussi les MCS pour calculer d'autres informations, comme le MUS qui 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] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In PASTE'05, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pages 82–87. ACM, 2005.

[3] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. In Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009.

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

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

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

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

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

[9] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher, and Antoine Rollet.

Constraint-based bmc : a backjumping strategy. STTT, 16(1) :103–121, 2014.

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

[11] 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.

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

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

[14] 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.

[15] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE, IEEE/ACM International Conference on Automated Software Engineering, pages 273–282. ACM, 2005.

[16] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In ICSE,Proceedings of the 22rd International Conference on Software Engineering, pages 467–477. ACM, 2002.

[17] 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.

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

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

[20] 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.

[21] 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.

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

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

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

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

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

[27] 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.

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

Annexe 2:

LocFaults: A new flow-driven and constraint-based error localization approach. .

Mohammed Bekkouche, Hélène Collavizza, Michel Rueher.

ACM SAC'15, SVT track, Apr 2015, Salamanca, Spain.

HAL : hal-01094227

LocFaults: A new flow-driven and constraint-based error localization approach*

Mohammed Bekkouche
University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
Mohammed.Bekkouche@i3s.unice.fr

Hélène Collavizza
University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
helen@polytech.unice.fr

Michel Rueher
University of Nice–Sophia Antipolis, I3S/CNRS
CS 40121, 06903 Sophia Antipolis Cedex, France
Michel.Rueher@unice.fr

ABSTRACT

We introduce in this paper LocFaults, a new flow-driven and constraint-based approach for error localization. The input is a faulty program for which a counter-example and a postcondition are provided. To identify helpful information for error location, we generate a constraint system for the paths of the 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. To compute the MCS, we extend the algorithm proposed by Liffiton and Sakallah [21] in order to handle programs with numerical statements more efficiently. The main advantage of this flow-driven approach is that the computed sets of suspicious instructions are small, each of them being associated with an identified path. Moreover, the constraint-programming based framework of LocFaults allows mixing Boolean and numerical constraints in an efficient and straightforward way. Preliminary experiments are quite encouraging.

*This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.
Part of this work was done while Michel Rueher was visiting professor at NII (National Institute of Informatics), Tokyo.

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2695822

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Constraints; D.2.5 [**Testing and Debugging**]: Debugging aids, Diagnostics, Error handling and recovery

General Terms

Verification, Algorithms

1. INTRODUCTION

Error localization from counter-examples and associated execution traces is a crucial issue in the software development process. Indeed, when a program P contains errors, a model checker usually provides a counter-example and an execution trace that is too long and too difficult to understand. This kind of outputs is therefore of limited interest for the programmer who has to debug the program. Thus, identifying code portions that may contain errors is often a difficult and expensive process, even for experienced programmers.

That is why we introduce here a new flow-driven and constraint-based approach for error localization. This new approach takes advantage from the structure of the Control Flow Graph (CFG) as well as from the flexibility provided by the constraint-programming framework. The process starts with a faulty program and a counter-example violating the postcondition. To provide helpful information for finding potential errors, we generate a constraint system for the paths of the CFG 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. In other words, we bound both the number of suspected assignments on the initial path, and the number of deviations from that path. To compute MCS, we extend the algorithms introduced by Liffiton and Sakallah [21, 20] to be able to handle programs with numerical statements more efficiently. Note that LocFaults may miss some errors since the number of deviations as well as the size of the MCS are bounded to limit combinatorial explosion.

To sum up, we take advantage of the information of the CFG for computing small sets of suspicious instructions, each of them being associated to an identified path. Moreover, this constraint-programming based framework provides an efficient and straightforward way for mixing Boolean and numerical constraints.

The rest of the paper is organized as follows. Section 2 illustrates how LocFaults works on a small example. Sec-

tion 3 goes into detail of the LocFaults framework. Section 4 reports experimental results on a number of benchmarks and problems, comparing our approach with BugAssist, a state-of-the art error localization framework [17, 18]. Section 5 discusses related work, summarizes the contributions and presents future research directions.

2. MOTIVATING EXAMPLE

Consider program `AbsMinus` (see fig. 1). The inputs are integers $\{i,\ j\}$ and the expected output is the absolute value of $i - j$. An error has been introduced in line 10, thus for the input data $\{i = 0, j = 1\}$, program `AbsMinus` returns -1. The postcondition here is just $result = |i - j|$[1].

```
1 class AbsMinus {
2 /*returns |i−j|,the absolute value of i minus j*/
3 /*@ ensures
4  @ ((i < j) ==> (\result == j−i)) &&
5  @ ((i >= j) ==> (\result == i−j)); */
6  int AbsMinus (int i, int j) {
7    int result;
8    int k = 0;
9    if (i <= j) {
10       k = k+2; } // error : k = k+2 instead of k=k+1
11   if (k == 1 && i != j) {
12       result = j-i; }
13    else {
14       result = i-j; }
15 return result;
  }
}
```

Figure 1: Program AbsMinus

The CFG of program `AbsMinus` and a faulty path are depicted in figure 2. This faulty path corresponds to the input data : $\{i = 0, j = 1\}$. First, LocFaults collects on path 2.(b) the constraint set $C_1 = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, r_1 = i_0 - j_0\}$[2]. Then, LocFaults computes the MCS of C_1. It is important to stress that the constraints defining the assignments of the input variables cannot belong to the computed MCS. Indeed, relaxing these constraints, rather corresponds to the generation of some test cases that satisfy the postcondition. So, only one MCS can be found in C_1: $\{r_1 = i_0 - j_0\}$. In other words, if we assume that the conditional statements are correct, the only suspicious statement on this faulty path is statement 14.

Then, LocFaults starts the deviation process. The first deviation (see figure 3.(a), green path) still produces a path that violates the postcondition. Thus, it is rejected. The second deviation (see figure 3.(b), blue path) produces a path that satisfies the postcondition. So, LocFaults collects the constraints on the part of path 3.(b) which precedes the deviated condition, that is $C_2 = \{i = 0, j = 1, k_0 = 0, k_1 = k_0 + 2\}$. Then LocFaults searches for an MCS of $C_2 \cup \neg(k = 1 \wedge i \neq j)$.That is to say, one tries to identify the assignments which must be modified to force the program to follow a path that satisfies the postcondition. Therefore, for this second deviation two suspicious statements are identified:

- The conditional statement in line 11;
- The assignment in line 10 since the corresponding constraint is the only MCS of $C_2 \cup \neg(k = 1 \wedge i \neq j)$.

Then, LocFaults goes on and tries to deviate a second condition. The only possible path is the one where both conditions of program `AbsMinus` are deviated. However, since it has the same prefix than the first deviated path, we discard it.

This example shows that LocFaults produces relevant and helpful information on each faulty path. Unlike to BugAssist, a state of art system, it does not merge all suspicious statements in a single set, which may be difficult to exploit by the user.

3. THE LocFaults FRAMEWORK

In this section we first introduce the formal definitions of MUS, MSS and MCS which are the basis of our error-localization framework. Then, we give an overview of BMC (Bounded Model Checking) based on constraint programming. Finally, we detail how we compute some MCS of bounded size along the program paths, using a depth-first search on the CFG of the program.

3.1 Error localization and MCS computation

Since we encode the set of statements of a faulty path as a set of constraints, the error localization problem is similar to the problem of finding a correction set for an inconsistent constraint system. This problem has been addressed both in the operational research community and constraint community. When searching useful information to correct inconsistent constraint systems, one may look for two kinds of information:

1. How many constraints in an unsatisfiable set of constraints can be satisfied ?
2. Which part of the constraint system is unsatisfiable ?

To answer these questions, the notion of MUS, MSS and MCS have been introduced by Liffiton and al [21].
A Minimal Unsatisfiable Subsets of constraints (MUS), also called "unsatisfiable cores" is an unsatisfiable system of constraints such that removing any one of its elements makes the remaining set of constraints satisfiable:

$$M \subseteq C \text{ is a MUS} \Leftrightarrow M \text{ is UNSAT}$$
$$\text{and } \forall c \in M : M \setminus \{c\} \text{ is SAT.}$$

A Maximal Satisfiable Subset (MSS) is a generalization of MaxSAT and MaxCSP where we consider the maximality instead of the maximum cardinality

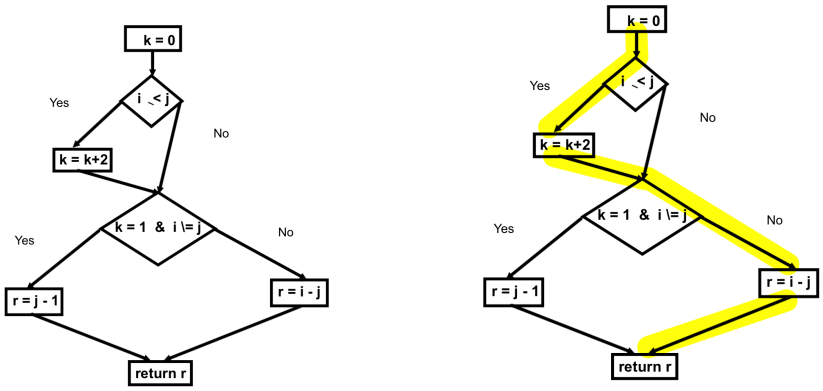
$$M \subseteq C \text{ is an MSS} \Leftrightarrow M \text{ is SAT}$$
$$\text{and } \forall c \in C \setminus M : M \cup \{c\} \text{ is UNSAT.}$$

The definition of the MSS is very close to that of IIS (Irreducible Inconsistent Subsystem) used in operational research [5, 6, 7].
An MCS is a subset of the constraints of an infeasible constraint system whose removal yields a satisfiable set of constraints ("correcting" the infeasibility). Furthermore it is minimal in the sense that any proper subset does not satisfy this property [21].

$$M \subseteq C \text{ is an MCS} \Leftrightarrow C \setminus M \text{ is SAT}$$

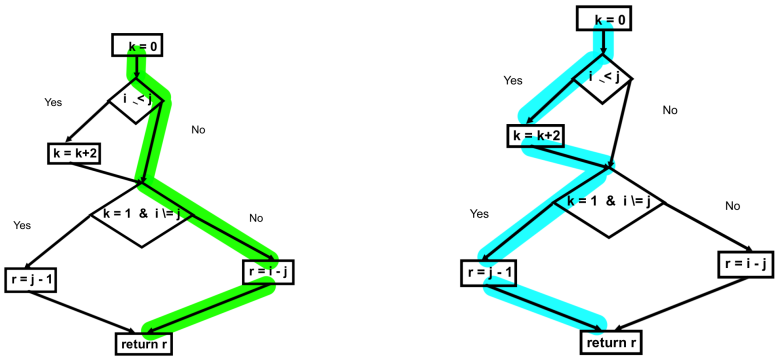
[1]Specifications are written in JML `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`

[2]Before collecting the constraints, a variable renaming is required. More precisely, the program has to be transformed in DSA form [2] which ensures that every variable is assigned at most once along any path.



(a) (b)

Figure 2: CFG and faulty path – Program `AbsMinus`



(a) (b)

Figure 3: Paths with one deviation – Program `AbsMinus`

$$\text{and } \forall c \in M : (C \setminus M) \cup \{c\} \text{ is UNSAT.}$$

There is a duality relationship between the set of MUS and the set of MCS [4, 21]

Different algorithms have been proposed for calculating IIS/MUS et MCS, e.g. `Deletion Filter`, `Additive Method`, `Additive Deletion Method`, `Elastic Filter` [5, 26, 6, 7]. Junker [19] proposed an algorithm based on a generic "Divide-and-Conquer" strategy to efficiently compute IIS or MUS when the cardinality of the conflict-subsets is much smaller than the one of the whole constraint set.

The algorithm introduced by Liffiton and Sakallah [21] first calculates all MCS in an increasing order, then all MUS by using the above property mentioned. The algorithm presented in section 3.4 is derived from that algorithm.

Various improvements of these algorithms [10, 20, 23] have been proposed during the last years but they are closely related to the specificity of SAT solvers, and thus, they are quite difficult to transpose to numeric constraint solvers.

Next section introduces the BMC framework we use to collect the constraints on a faulty or suspicious path.

3.2 Constraint based BMC

Constraint based BMC is a BMC approach that uses constraints for modeling the program and its specification, and constraint solving for checking if the program conforms its specification [8, 9]. BMC uses a bound b to unfold loops: they are replaced with conditional statements of depth b. If an error is found for a given b, then the program does not conform its specification. Otherwise, b is increased for searching a deeper error, until a maximum value of b has been reached.

Let $PROG_b$ be the program after being unfolded b times. One BMC step checks the Hoare triplet $\{PRE, PROG_b, POST\}$ where PRE is the precondition and $POST$ is the postcondition. $PROG_b$ does not conform its specification if the formula $\Phi = PRE \wedge PROG_b \wedge \neg POST$ is satisfiable. An instantiation of the variables of Φ is then a *counter-example* because it satisfies both the precondition and the program, but it does not satisfy the postcondition.

CPBPV [8] is a BMC tool based on constraint programming. $CPBPV$ translates PRE and $POST$ into constraints, and transforms $PROG_b$ into a CFG whose nodes are the conditions and assignments of the program translated into constraints[3]. CPBPV builds the constraint system CSP associated to formula Φ *on the fly*, using a depth-first search on the data structure of the graph. At the initial state, CSP contains the constraints based from PRE and $\neg POST$. Then the constraints of a path are added during the graph exploration. When the last node has been reached on a path, the satisfiability of CSP is checked. When CSP has a solution, an error has been found in the program thus the BMC process is stopped. Otherwise, another branch is explored. When all the branches have been explored without finding

[3]To avoid the problem of multiple definition of variables, we use the DSA (Dynamic Single Assignment) form [2].

any solution, then $PROG_b$ is conform with its specification.

3.3 MCSs on a path

In this paper, we extend our BMC approach to *locate* which part of the program may be responsible for the error found during the BMC step. More precisely, let CE be a counter-example found during the BMC step. CE is the solution of CSP which contains the constraints from the precondition, the negation of the postcondition, and the constraints based on the assignments collected on the faulty path. Let $PATH$ denote this last set of constraints. Then the constraint system $C_{path} = CE \cup PRE \cup PATH \cup POST$ is ***UN**satisfiable* since CE is a counter-example that satisfies $\neg\ POST$. An MCS of C_{path} is a set of constraints which must be removed in order to make C_{path} satisfiable. By definition, such an MCS is a possible *error localization* on the faulty path. This first localization step assumes that the error is an assignment on the counter-example path. But the program can also be wrong because of a bad choice on a conditional node. LocFaults also computes bonded MCS on paths which are built by changing some conditions on the initial faulty path.

3.4 Algorithm scheme

The inputs of our algorithm are the CFG of the program, CE the counter-example, b_{cond}, a bound on the number of conditions which are diverted and b_{mcs} a bound on the number of MCS which are computed on each path. CE is a set of values of the input variables of the program. Roughly speaking, our algorithm traverses the CFG using CE to select one or the other branch of each conditional node, and collects the constraints associated with the assignments on the induced path. It changes zero, one or at most b_{cond} decisions on this path. At the end of a path, the set of constraints which have been collected is unsatisfiable, and at most b_{mcs} MCSs are computed on this CSP.

More precisely LocFaults proceeds as follows :

- It first propagates CE on the CFG until the end of the faulty path has been reached. Then it computes atmost b_{mcs} MCSs on the current CSP. This is a first localization on the counter-example path.
- Then LocFaults tries to divert one condition. When the first conditional node *cond* is reached, LocFaults takes the opposite decision as the one induced by CE, and continues to propagate CE to the last CFG node. If the CSP built from this diverted path is satisfiable, there are two kinds of *suspicious error set* :
  - the first one is the condition *cond* itself. Indeed, changing the decision for *cond* makes CE satisfies the postcondition $POST$,
  - another possible cause of the error is that a bad assignment before *cond* had produced a wrong decision. Thus LocFaults also computes atmost b_{mcs} MCSs on the CSP that contains the constraints collected on the path that reaches *cond*.

  This process is repeated on each conditional node of the counter-example path.
- A similar process is then applied for diverting k conditions for all $k \leq k_{max}$. To increase efficiency, the conditional nodes which correct the program are marked with the number of diversions which have been made before they had been reached. For a given step k, if changing the decision of a conditional node *cond* marked with value k' with $k' \leq k$ corrects the program, this correction is ignored. In other words, we only consider the first time where a conditional node corrects the program.

4. EXPERIMENTS

To evaluate the capabilities of our approach we experimented with two sets of Benchmarks : the well known TCAS suite from Siemens[25], and a set of variations of the `Tritype` program. We compared the results of LocFaults with the one of BugAssist.

TCAS is an aircraft collision avoidance system. The program contains 173 lines of C code with almost no arithmetic operations. The suite contains 41 faulty versions .

The `Tritype` program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths because of complex conditional statements in the program. The program takes three positive integers as inputs (i, j, k) the triangle sides, and returns the value 2 if the inputs correspond to an isosceles triangle, the value 3 if they correspond to an equilateral triangle, the value 1 if they correspond to some other triangle, and the value 4 otherwise. The `Tritype` program is also a decision problem but we derived from the initial program two versions with more arithmetic operations. The first returns the product of the length of the sides, whereas the second one computes the square of the surface of the triangle by using Heron's formula.

The results of these experiments are detailed in the next subsections. All experiments were done on an Intel Core Core i7-3720QM at 2.6 GHz with 8 GB of memory running 64-bit Linux. LocFaults uses the IBM solvers CP OPTIMIZER and CPLEX [4].

4.1 TCAS suite

The results of the experiments are reported on Table 1. First column specifies the TCAS version number, the second one the number of errors in the program and the third column gives the number of the generated counter-examples. The two last columns provide the number of errors that have been found by LocFaults and BugAssist. Some of the versions are omitted because the errors correspond to array index out of bound and we still cannot handle this kind of overflow errors. We didn't report the computation times because there is no significant difference. The reported results for LocFaults have been obtained with at most one deviation; except for version `V41` where two deviations were required.

The size of the set of suspicious instructions identified by BugAssist is in general larger than the sum of the sizes of the sets of suspicious instructions generated by LocFaults but BugAssist identifies a bit more errors than LocFaults. More importantly, since LocFaults reports a set of MCS for each faulty path, the error localization process is much more easier than with the single set of suspicious errors reported by BugAssist.

In all, the performances of LocFaults and BugAssist

[4] `http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/`

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

Table 1: Results on TCAS

are very similar on this benchmark well adapted for a Boolean solver.

4.2 Variations on the `Tritype` program

The results of the experiments on the different variations of the `Tritype` program[5] are reported in Table 2. In that table, the numbers are the line numbers and the red numbers are the injected errors that have been found by the tools. For LocFaults, the underlined numbers are conditions, one line corresponds to a path through the CFG, and contains either a condition alone, or condition and the assignments before that condition which allow to change the branch. For example, for `TritypeV1` in column = 1 where one condition is diverted, LocFaults first locates the condition 26 as being erroneous. Then it locates condition 48 and locates the assignments 30 or 25 which can be responsible of the bad decision on 48.

Versions 1 to 5 of `Tritype` correspond to the standard `Tritype` program where we injected different kinds of errors.

- `TritypeV1` : the error was introduced in the last assignment statement of the program. LocFaults identified this error in the first step, without deviating any condition.
- `TritypeV2` : the error is in a nested condition, just before the last assignment. LocFaults finds the relevant suspicious statement after 4 deviations. BugAssist identifies also the relevant suspicious statement.
- `TritypeV3` : the error is an assignment and will entail a bad branching. Here again, LocFaults only finds it after 4 deviations but all suspicious set contains only one statement.
- `TritypeV4`: the error is in a condition, at the beginning of the program. LocFaults finds it very quickly. Even the first identified suspicious statement may be helpful : it is an assignment, just after the wrong condition.
- `TritypeV5` : there are two wrong conditions in this program. LocFaults needs to divert 3 conditions to find the two errors, while BugAssist only finds the first one.
- `TritypeV6` : is a variation that returns the perimeter of the triangle. LocFaults identified this error in the first step, without deviating any condition.

Versions 7 and 8 of `Tritype` are some variations of the original program that return *non linear* expressions. They have the same control structure as `Tritype`. `TritypeV7` computes the product of the three sides and `TritypeV8` computes the square of the area of the triangle. The specification of `TritypeV8` uses the Heron formula $\sqrt{s(s-i)(s-j)(s-k)}$ where $s = (i+j+k)/2$. To ensure that the returned value is an integer, we compute the square of the area and we assume as precondition that s is even. Moreover, the returned value varies according to the triangle type. For example, if the triangle is isosceles and $i == j$, the returned value is $s(s-i)(s-i)(s-k)$, and if the triangle is equilateral, the returned value is $(3 \times i^4)/16$.

Computations times are very short for all programs but `TritypV7` and `TritypV8`. Table 3 reports the times for these two programs. P stands for the pre-processing time[6] whereas the other rows contain the solving time. LocFaults is an order of magnitude faster than BugAssist on these two benchmarks. This clearly shows the benefit of using a constraint solver for programs containing non-trivial numerical statements. Indeed, these numerical constraints are much more difficult to handle by the SAT solver used in BugAssist than by the CSP solver used in LocFaults.

On all these benchmarks, the size of the set of suspicious instructions identified by BugAssist is similar to the sum of the sizes of the sets of suspicious instructions generated by LocFaults. But these benchmarks also show that the debugging process is much easier with the small set provided by LocFaults than with the global set of suspicious instructions computed by BugAssist. Our approach is a flow-based approach, that generates the sets of suspicious instructions in an incremental way. It finds errors along the path of the counter-example, and reports some explanations in an order that can help the user to find the bug. This is more appropriate for debugging than a global approach like BugAssist which computes a single set of suspicious instructions.

[5]The source code of these benchmarks can be found at: `http://users.polytech.unice.fr/~rueher/Benchs/LocF/`

[6]For LocFaults this is the time needed for the JDT Eclipse parser to build the AST from the Java program.

Program	Counter-example	Errors	LocFaults				BugAssist
			= 0	= 1	= 2	= 3	
TritypeV1	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48},{30},{25}	{29, 32} {53, 57},{30},{25}	/	{26, 27, 32, 33, 36, 48, 57, 68}
TritypeV2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{27},{25}	{29, 57} {32, 44}	/	{21, 26, 27, 29, 30, 32, 33, 35, 36, 33, 35, 36, 53, 68}
TritypeV3	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25}	{33, 45}	/	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
TritypeV4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45},{33},{25}	{26, 32}	{32, 35, 49} {32, 35, 53} {32, 35, 57}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TritypeV5	$\{i = 2, j = 3, k = 3\}$	32,45	{40}	{26} {29}	{32, 45} {35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TritypeV6	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}
TritypeV7	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{27},{32}	/	/	{72, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62}
TritypeV8	$\{i = 3, j = 4, k = 3\}$	61	{61}	{29} {35},{30},{25}	/	/	{19, 61, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 54}

Table 2: Tritype benchmark

Programme	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
TritypeV7	$0,722s$	$0,051s$	$0,112s$	$0,119s$	$0,144s$	$0,140s$	$20,373s$
TritypeV8	$0,731s$	$0,08s$	$0,143s$	$0,156s$	$0,162s$	$0,216s$	$25,562s$

Table 3: Computation times for non linear programs

5. DISCUSSION

5.1 Related work

Various techniques to support error localisation have been proposed in the test and verification community.

The problem of error localization was first addressed in the test community where many systems have been developed. The most famous one is Tarantula [16, 15] that uses different metrics to rank suspicious statements detected while running a battery of tests. The critical point of this approach is that it requires an oracle for deciding whether the test result is correct or not. To avoid this problem, we consider here the Bounded-Model Checking (BMC) framework where we only require a postcondition or an assertion to check.

In this BMC context Bal et al [1] developed one of the first error localization system. Roughly speaking, they perform multiple calls to a model checker and compare the trace of generated counter-examples with a correct execution trace. Transitions that are not included in the correct trace are reported as a possible cause of the error. Their algorithm has been implemented in the of SLAM BMC framework that verifies temporal safety properties of C programs.

More recently, approaches based on the derivation of correct traces were introduced in Explain[14, 13]. Explain works as follows:

1. Calling the model-checker CBMC[7] to find an execution that violates the postcondition;
2. Using a pseudo-Boolean solver for searching the nearest correct execution;
3. Computing the difference between both traces.

Then, Explain produces a propositional formula S associated with program P but whose assignments do not violate the specification. Finally, Explain extends S with constraints defining an optimization problem the goal of which is to find a satisfying assignment that is as close as possible to the counter example; proximity is measured by a distance on the execution of P.

An approach similar to one of Explain was introduced in [24] but it is based on testing rather than on model checking: the authors use a series of correct and incorrect tests and a distance metrics to select a correct test from a given set of test data. This approach assumes that a full oracle is available.

In [11, 12], the authors also start from a counter-example, but they use the specification to derive a correct program for the same input data. Each identified statement can be used to correct errors. This approach ensures that the errors are inside the set of suspicious statements (assuming that the error is in the considered erroneous model). In other words, their approach identifies a super set of erroneous statements. To reduce the number of potential errors, the process is restarted for various counter-examples and the intersection of the sets of suspicious statements are calculated. However, this approach suffers from two major problems:

- The search space may be very large since every expression can be modified;
- It generates numerous spurious diagnoses since any change in an expression is possible (for example, chang-

[7] http://www.cprover.org/cbmc/

ing the last assignment of a function to return the expected result).

To overcome these problems, Zhang et al [28] proposed to change only predicates of the control flow. The intuition behind this approach is that by switching the results of a predicate and modifying the control flow, the program state cannot only be inexpensively modified, but in addition, it is often possible to reach a successful state. Liu et al [22] generalized this approach by editing multiple predicates. They also propose a theoretical study of debugging algorithm for *RHS* errors, that is, errors in predicate control and in the right hand side of assignments.

In [3], the authors address the problem of analyzing the trace of a counter-example and of error localization in the context of formal verification of hardware systems. They use the notion of causality introduced by Halpern and Pearl to formally define a set of causes of the violation of the specification by a counter-example.

Manu Jose and Rupak Majumdar [17, 18] have addressed this problem differently: they introduced a new algorithm that uses a MAX-SAT solver to calculate the maximum number of clauses of a Boolean formula that can be satisfied by an assignment. Their algorithm works in three steps:

1. They encode a trace of a program by a Boolean formula F that is satisfiable if and only if the trace is satisfiable;

2. They build a false formula F' by requiring that the postcondition is true (the formula F' is unsatisfiable because the trace models a counter-example that violates the postcondition);

3. They use MAXSAT to compute the maximum number of clauses that can be satisfied in F' and display the complement of this set as a potential cause of errors. In other words, they calculate the complement of a MSS (Maximum Satisfiable Subset).

Manu Jose and Rupak Majumdar [17, 18] have implemented their algorithm in BugAssist, a state-of-art verification tool based on CBMC.

Si-Mohamed Lamraoui and Shin have recently developed SNIPER, a tool that generalizes the approach of BugAssist in order to improve error localization in programs with multiples faults. SNIPER calculates the MSS of a formula $\psi = EI \wedge TF \wedge AS$ where EI encodes the erroneous input values, TF denotes a formula modelling all paths of the program, and AS is the violated assertion. The MCSs are obtained by taking the complement of the calculated MSS. The implementation is based on the intermediate representation LLVM and the SMT solver Yices. The authors compared SNIPER with BugAssist on the Siemens test suite TCAS. SNIPER identified by anywhere 5% more errors than BugAssist but required about much more times than BugAssist on this benchmark. More importantly, their approach assumes that a set of inputs that triggers all faults in the program is available.

In [27], the authors propose to compute irreducible infeasible subsets of constraints. They use a constraint solver to derive the MCS from these sets but the number of single fault candidates they generate is rather large.

There are some similarities between the approach we propose here and the framework introduced by Manu Jose and Rupak Majumdar. The main differences are:

1. We use the control flow graph to collect the constraints on the path from the counter example, as well as on the derived paths from the path of the counter-example by assuming that at most k conditional statements may contain errors. So, we do not transform the whole program into a system of constraints.

2. We use more general algorithms than MAXSAT that make it easier to deal with numerical constraints.

5.2 Contribution and future work

Our flow-based and incremental approach is a good way to help the programmer with bug hunting since it locates the errors around the path of the counter-example. Further work concerns programs with loops where scalability may be an issue. We plan also to develop an interactive version of our tool that provides the localizations one after the others, and takes benefit from the user knowledge to select the condition that must be diverted.

Furthermore, the constraint-based framework that we have introduced is well adapted for handling arithmetic operations. Moreover, it can be extended in straightforward way for error-localization in programs with floating-point numbers computations. Such an extension would be more difficult in SAT-based framework of BugAssist, or in SNIPER where the whole program is transformed in anLLVM intermediate representation.

6. REFERENCES

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

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In PASTE'05, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pages 82–87. ACM, 2005.

[3] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining counterexamples using causality. In Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009.

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

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

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

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

[8] H. Collavizza, M. Rueher, and P. V. Hentenryck. Cpbpv: a constraint-programming framework for bounded program verification. Constraints, 15(2):238–264, 2010.

[9] H. Collavizza, N. L. Vinh, O. Ponsini, M. Rueher, and A. Rollet. Constraint-based bmc: a backjumping strategy. STTT, 16(1):103–121, 2014.

[10] A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM, 26(1):53–62, 2012.

[11] A. Griesmayer, R. Bloem, and B. 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.

[12] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for c programs. Electr. Notes Theor. Comput. Sci., 174(4):95–111, 2007.

[13] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. STTT, 8(3):229–247, 2006.

[14] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In Proceedings of CAV, volume 3114 of Lecture Notes in Computer Science, pages 453–456. Springer, 2004.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE, IEEE/ACM International Conference on Automated Software Engineering, pages 273–282. ACM, 2005.

[16] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In ICSE,Proceedings of the 22rd International Conference on Software Engineering, pages 467–477. ACM, 2002.

[17] M. Jose and R. 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.

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

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

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

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

[22] Y. Liu and B. Li. Automated program debugging via multiple predicate switching. In Proceedings of AAAI. AAAI Press, 2010.

[23] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013.

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

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

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

[27] F. Wotawa, M. Nica, and I. Moraru. Automated debugging based on a constraint model of the program and a test case. J. Log. Algebr. Program., 81(4):390–407, 2012.

[28] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006.