Localisation de fautes à l'aide de la fouille de données sous contraintes

Mehdi Maamar, Nadjib Lazaar, Samir Loudni, Yahia Lebbah

Localisation de fautes à l'aide de la fouille de données sous contraintes

Mehdi Maamar[1], Nadjib Lazaar[2], Samir Loudni[3], Yahia Lebbah[1]

[1] Université d'Oran 1, Lab. LITIO, BP 1524, El-M'Naouer, 31000 Oran, Algérie.
[2] CNRS, Université de Montpellier, LIRMM, 161 rue Ada, 34090 Montpellier, France
[3] Université de Caen, CNRS, UMR 6072 GREYC, 14032 Caen, France.

Résumé Nous proposons dans cet article une approche basée sur la fouille de motifs ensemblistes sous contraintes pour la localisation des fautes dans les programmes. Nous formalisons le problème de localisation des fautes comme un problème d'extraction des k meilleurs motifs satisfaisant un ensemble de contraintes modélisant les instructions les plus suspectes. Nous faisons appel à la programmation par contraintes pour modéliser et résoudre le problème de localisation. Les expérimentations menées sur une série de programmes montrent que notre approche offre une localisation plus précise comparée à Tarantula [10].

1 Introduction

L'aide à la localisation d'erreurs à partir 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 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. Au cours de la dernière décennie, plusieurs techniques automatisées ont été proposées pour résoudre ce problème.

La plupart de ces techniques automatisées comparent deux types de traces d'exécution, les exécutions correctes (dites positives) et les exécutions erronées (dites négatives) [10,11]. Ces méthodes se basent sur une fonction de score pour évaluer le caractère suspect de chaque instruction dans le programme en exploitant les occurrences des instructions apparaissant dans les traces négatives et positives. L'intuition sous-jacente est que les instructions ayant les scores les plus élevés sont les plus suspectes pour contenir des fautes. Toutefois, l'inconvénient majeur de ces techniques est que les dépendances entre les instructions sont ignorées. Des techniques ont été proposées afin de prendre en compte les chaines de cause à effet avec une analyse des dépendances en utilisant, par exemple, le découpage de programmes [1].

En outre, ces dernières années, le problème de localisation de fautes a été traité avec des techniques de fouille de données. Cellier et al. [2,3] proposent une approche basée sur les règles d'associations et l'analyse formelle de concepts

(FCA) pour assister la localisation de fautes. Nessa et al. [14] proposent de générer des sous-séquences d'instructions de longueur N, appelées *N-grams*, à partir des traces. Les traces d'exécution erronées sont examinées pour trouver les N-grams avec un taux d'occurrence supérieur à un certain seuil. Une analyse statistique est ensuite conduite pour déterminer la probabilité conditionnelle qu'une exécution échoue sachant qu'un certain N-gram apparaît dans sa trace.

D'autre part, plusieurs travaux en fouille de données ont mis en avant l'intérêt d'utiliser les contraintes pour indiquer le type de motifs à extraire afin de cibler le processus d'extraction suivant les centres d'intérêt de l'utilisateur. Comme le procédé produit en général un grand nombre de motifs, un grand effort est fait pour mieux comprendre l'information fragmentée véhiculée par les motifs et produire des sous-ensembles de motifs satisfaisant des propriétés sur l'ensemble de tous les motifs [5,16]. L'extraction des top-k motifs (i.e. les k meilleurs motifs selon une fonction de score) est une voie récente en fouille de données sous-contraintes permettant de produire des motifs intéressants [4].

Dans ce papier, nous proposons une approche basée sur la fouille de motifs ensemblistes pour la localisation de fautes. Nous formalisons le problème de localisation de fautes comme un problème d'extraction des k meilleurs motifs satisfaisant un ensemble de contraintes modélisant les instructions suspectes. Notre approche bénéficie des travaux récents sur la fertilisation croisée entre la fouille de données et la programmation par contraintes [8,12]. L'approche comporte deux étapes : i) extraction des top-k suites d'instructions les plus suspectes et ii) classement des instructions issues des top-k motifs pour localiser la faute.

Les expérimentations menées sur une série de programmes Siemens montrent que notre approche permet de proposer une localisation plus précise comparée à la technique de localisation de fautes la plus populaire (Tarantula [10]).

L'article est organisé comme suit. La section 2 introduit les concepts de base. La section 3 présente les principales notions liées à l'extraction de motifs. La section 4 décrit notre approche pour la localisation de fautes. Les résultats de nos expérimentations sont donnés en section 5.

2 Définitions et motivations

Cette section introduit les notions et les définitions habituellement utilisées en localisation de fautes et en satisfaction de contraintes.

2.1 Problème de localisation de fautes

En génie logiciel, une défaillance est une déviation entre le résultat attendu et le résultat actuel. Une erreur est une partie du programme qui est susceptible de conduire à une défaillance. Une faute est la cause supposée ou adjugée d'une erreur [13]. L'objectif de la localisation de fautes est d'identifier la cause principale des symptômes observés dans les cas de test.

Soit un programme P ayant n lignes $L = \{e_1, e_2, ..., e_n\}$. La figure 1 montre un exemple de programme "compteur de caractères", obtenu après suppression

Programme : Compteur de caractères	Cas de test tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8
`function count (char *s) {`								
`int let, dig, other, i = 0;`								
`char c;`								
e_1 : `while (c = s[i++]) {`	1	1	1	1	1	1	1	1
e_2 : `if('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	0	1
e_3 : `let += 2; //- faute -`	1	1	1	1	1	1	0	0
e_4 : `else if ( 'a'<=c && 'z'>=c )`	1	1	1	1	1	0	0	1
e_5 : `let += 1;`	1	1	0	0	1	0	0	0
e_6 : `else if ( '0'<=c && '9'>=c )`	1	1	1	1	0	0	0	1
e_7 : `dig += 1;`	0	1	0	1	0	0	0	0
e_8 : `else if (isprint (c))`	1	0	1	0	0	0	0	1
e_9 : `other += 1;`	1	0	1	0	0	0	0	1
e_{10} : `printf("%d %d %d\n", let, dig, other);}`	1	1	1	1	1	1	1	1
Passing/Failing	F	F	F	F	F	F	P	P

FIGURE 1: Exemple d'un programme avec sa matrice de couverture.

des commentaires et des instructions non-exécutables comme les déclarations de variables et de fonctions, avec $L = \{e_1, e_2, ..., e_{10}\}$.

Un cas de test tc_i est un tuple $\langle D_i, O_i \rangle$, où D_i est l'ensemble des paramètres d'entrée pour déterminer si un programme P se comporte comme prévu ou non, et O_i est la sortie attendue du programme. Soit $\langle D_i, O_i \rangle$ un cas de test et A_i la sortie courante retournée par un programme P suite à l'exécution de l'entrée D_i. Si $A_i = O_i$, alors le cas de test tc_i est considéré comme *réussi* (i.e. positif), *erroné* (i.e. négatif) dans le cas contraire. Une suite de tests $T = \{tc_1, tc_2, ..., tc_m\}$ est l'ensemble des m cas de test destinés à tester si le programme P respecte l'ensemble des exigences spécifiées. Étant donné un cas de test tc_i et un programme P, l'ensemble des instructions de P exécutées (au moins une fois) avec tc_i désigne la couverture du cas de test $I_i = (I_{i,1}, ..., I_{i,n})$, où $I_{i,j} = 1$ si l'instruction à la ligne j est exécutée, 0 sinon. La couverture d'un cas de test indique quelles sont les parties actives du programme durant une exécution spécifique. Par exemple, le cas de test tc_4 dans la figure 1 couvre les instructions $(e_1, e_2, e_3, e_4, e_6, e_7, e_{10})$. Le vecteur de couverture correspondant est $I_4 = (1, 1, 1, 1, 0, 1, 1, 0, 0, 1)$.

2.2 Problèmes de Satisfaction de Contraintes

Un CSP est un triplet $(X, \mathcal{D}, \mathcal{C})$ tel que $X = \{X_1, \ldots, X_n\}$ est l'ensemble fini des variables. $\mathcal{D} = \{D_1, \ldots, D_n\}$ est l'ensemble des domaines des variables, où chaque D_i est un ensemble fini contenant les valeurs possibles pour la variable X_i. $\mathcal{C} = \{C_1, \ldots, C_e\}$ est l'ensemble des contraintes $\mathcal{C}$. Chaque contrainte C_i porte sur un sous-ensemble de variables de X appelé $var(C_i)$. Le but est de trouver une instanciation $(X_i = d_i)$ avec $d_i \in D_i$ pour $i = 1, \ldots, n$, tel que toutes les contraintes soient satisfaites.

3 Extraction de motifs ensemblistes

Cette section présente les principales notions liées à l'extraction de motifs ainsi que la modélisation, sous forme de CSP, des problèmes d'extraction de motifs sous contraintes [8].

3.1 Contexte et définitions

Soit $\mathcal{I}$ un ensemble de littéraux distincts appelés *items* et $\mathcal{T} = \{1, ..., m\}$ un ensemble d'identifiants de transactions. Un motif ensembliste d'items est un sous-ensemble non vide de $\mathcal{I}$. Ces motifs sont regroupés dans le langage $\mathcal{L}_\mathcal{I} = 2^\mathcal{I} \backslash \emptyset$. Un jeu de données transactionnel est l'ensemble $\mathbf{r} \subseteq \mathcal{I} \times \mathcal{T}$.

Definition 1 (Couverture et fréquence). *La couverture d'un motif x est l'ensemble de tous les identifiants de transactions qui le supportent : $couv_\mathcal{T}(x) = \{t \in \mathcal{T} | \forall i \in x, (i,t) \in r\}$. La fréquence d'un motif x représente le cardinal de sa couverture $freq_\mathcal{T}(x) = |couv_\mathcal{T}(x)|$.*

L'extraction de motifs sous contraintes consiste à extraire les motifs satisfaisant une contrainte C à partir d'un jeu de données $\mathbf{r}$. La contrainte de *motifs fréquents* est définie en sélectionnant les motifs dont la fréquence est supérieure à un seuil donné min_{fr}. La contrainte de *taille* contraint le nombre d'items d'un motif x.

Une limite bien connue de l'extraction de motifs est le nombre de motifs produits qui peut être très grand. Les *représentations condensées* de motifs satisfaisant une contrainte C ont été introduites pour augmenter la rapidité d'exécution des algorithmes d'extraction de motifs et réduire le nombre de motifs obtenus. Pour les motifs ensemblistes et la contrainte de fréquence minimale, la représentation la plus usuelle est celle fondée sur les *motifs fermés*.

Definition 2 (Motif fermé). *Un motif $x \in \mathcal{L}_\mathcal{I}$ est fermé par rapport à la fréquence ssi $\forall\, y \supsetneq x, freq(y) < freq(x)$.*

Les motifs fermés structurent le treillis des motifs en *classes d'équivalence*. Tous les motifs d'une même classe d'équivalence ont la même couverture. Les motifs fermés correspondent aux éléments maximaux (au sens de la taille des motifs) des classes d'équivalence.

Par ailleurs, l'utilisateur est souvent intéressé par la découverte de motifs plus riches satisfaisant des contraintes portant sur un ensemble de motifs et non plus un seul motif. Ces contraintes sont définies comme étant des contraintes sur des *ensembles de motifs* [5] ou sur des motifs *n-aires* [12]. L'approche que nous proposons dans ce papier permet de traiter ce type de motifs tel que les top-k motifs.

Definition 3 (top-k motifs). *Soit m une mesure d'intérêt et k un entier. Les top-k motifs est l'ensemble des k meilleurs motifs par rapport à la mesure m :*

$$\{x \in \mathcal{L}_\mathcal{I} \mid freq_\mathcal{T}(x) \geq 1 \wedge \nexists y_1, \ldots, y_k \in \mathcal{L}_\mathcal{I} : \forall 1 \leq j \leq k, m(y_j) > m(x)\}$$

3.2 Modélisation sous forme d'un CSP

Soit $\mathbf{r}$ un jeu de données ou $\mathcal{I}$ est l'ensemble de ses n items et $\mathcal{T}$ l'ensemble de ses m transactions. $\mathbf{r}$ peut être représenté par une matrice booléenne $\mathbf{d} = (d_{t,i})_{t \in \mathcal{T}, i \in \mathcal{I}}$, tel que $\forall t \in \mathcal{T}, \forall i \in \mathcal{I}$, $(d_{t,i} = 1) \Leftrightarrow (i \in t)$. Soit M le motif

recherché. M est représenté par n variables booléennes $\{X_1, X_2, ..., X_n\}$ tel que : $\forall i \in \mathcal{I},\ (X_i = 1)\ ssi\ (i \in M)$. Le support du motif recherché M est représenté par m variables booléennes $\{T_1, T_2, \ldots, T_m\}$ tel que : $\forall t \in \mathcal{T},\ (T_t = 1)\ ssi\ (M \subseteq t)$. La relation entre le motif recherché M, son support et le jeu de données $\mathbf{r}$ est établie par des contraintes réifiées imposant que, pour chaque transaction t, $(T_t = 1)$ ssi M est un sous ensemble de t [8] :

$$\forall t \in \mathcal{T}, (T_t = 1) \leftrightarrow \sum_{i \in \mathcal{I}} X_i \times (1 - d_{t,i}) = 0 \tag{1}$$

L'encodage booléen permet d'exprimer de façon simple certaines mesures usuelles : $freq(M) = \sum_{t \in \mathcal{T}} T_t$ et $taille(M) = \sum_{i \in \mathcal{I}} X_i$. La contrainte de fréquence minimale $freq(M) \geq min_{fr}$ (ou min_{fr} est un seuil) est encodée par la contrainte $\sum_{t \in \mathcal{T}} T_t \geq min_{fr}$. De la même manière, la contrainte de taille minimale $taille(M) \geq \alpha$ (ou α est un seuil) est encodée par la contrainte $\sum_{i \in \mathcal{I}} X_i \geq \alpha$.

Finalement, la contrainte de fermeture $fermé_m(M)$ (Avec $m = freq$) est encodée par l'équation (2).

$$\forall i \in \mathcal{I}, (X_i = 1) \Leftrightarrow \sum_{t \in \mathcal{T}} T_t \times (1 - d_{t,i}) = 0 \tag{2}$$

4 Localisation de fautes par fouille d'ensemble de motifs

Cette section montre notre formalisation du problème de localisation de fautes comme un problème d'extraction des k meilleurs motifs satisfaisant un ensemble de contraintes qui modélisent les instructions les plus suspectes d'un programme.

4.1 Encodage booléen

Soit $L = \{e_1, ..., e_n\}$ l'ensemble des instructions composant le programme P et $T = \{tc_1, ..., tc_m\}$ l'ensemble des cas de test. La base transactionnelle $\mathbf{r}$ est définie comme suit : (i) chaque instruction de L correspond à un item de $\mathcal{I}$, (ii) la couverture de chaque cas de test forme une transaction dans $\mathcal{T}$. Par ailleurs, pour découvrir les contrastes entre les sous-ensembles de transactions, $\mathcal{T}$ est partitionné en deux sous-ensembles disjoints $\mathcal{T}^+$ et $\mathcal{T}^-$. $\mathcal{T}^+$ (resp. $\mathcal{T}^-$) représente l'ensemble des couvertures des cas de test positifs (resp. négatifs).

Soit $\mathbf{d}$ la matrice booléenne représentant la base transactionnelle $\mathbf{r}$. Nous avons alors, $\forall t \in \mathcal{T}, \forall i \in \mathcal{I},\ (d_{t,i} = 1)$ si et seulement si l'instruction i est exécutée par le cas de test t. La figure 1 montre la table des transactions associée au programme "compteur de caractères". Par exemple, la couverture du cas de test tc_5 est $I_5 = (1, 1, 1, 1, 1, 0, 0, 0, 0, 1)$. Comme tc_5 est un cas de test qui a échoué, alors $I_5 \in \mathcal{T}^-$.

Soit M le motif recherché (i.e. suite d'instructions). Comme dans [6], nous introduisons deux ensembles de variables booléennes : n variables $\{X_1, ..., X_n\}$

pour représenter le motif M et m variables $\{T_1, ..., T_m\}$ pour représenter le support de M. Comme pour $\mathcal{T}$, l'ensemble $\{T_1, ..., T_m\}$ est partitionné en deux sous-ensembles disjoints : les variables $\{T_t^+\}$ représentant le support de M dans $\mathcal{T}^+$ et les variables $\{T_t^-\}$ représentant le support de M dans $\mathcal{T}^-$.

Afin de réduire la redondance dans les suites d'instructions extraites, nous imposons la contrainte de fermeture $fermé_m(M)$ $(m = freq)$ (cf. équation [4] (3)) :

$$\forall i \in \mathcal{I}, (X_i = 1) \Leftrightarrow \left(\sum_{t \in \mathcal{T}^+} T_t^+ \times (1 - d_{t,i}^+) = 0 \right) \wedge \left(\sum_{t \in \mathcal{T}^-} T_t^- \times (1 - d_{t,i}^-) = 0 \right) \quad (3)$$

4.2 Extraction des top-k motifs les plus suspects

L'intuition qui est derrière la grande majorité des méthodes de localisation de fautes est que les instructions qui apparaissent dans les traces d'exécution négatives sont les plus suspectes, tant dis que les instructions qui apparaissent seulement dans les traces d'exécutions positives sont les plus innocentes [7,10,14,15].

Pour extraire les motifs les plus suspects, nous définissons une relation de dominance $\succ_{\mathcal{R}}$ entre les motifs en fonction de leur niveau de suspicion.

Definition 4 (Relation de dominance). *Étant donné une bipartition de $\mathcal{T}$ en deux sous-ensembles disjoints $\mathcal{T}^+$ et $\mathcal{T}^-$, un motif M domine un autre motif M' (noté $M \succ_{\mathcal{R}} M'$), ssi*

$$\begin{aligned}\left[freq_{\mathcal{T}^-}(M) > freq_{\mathcal{T}^-}(M')\right] \vee [(freq_{\mathcal{T}^-}(M) = freq_{\mathcal{T}^-}(M')) \wedge \\ (freq_{\mathcal{T}^+}(M) < freq_{\mathcal{T}^+}(M'))]\end{aligned} \quad (4)$$

La relation de dominance exprime le fait que M sera préféré à M', noté $M \succ_{\mathcal{R}} M'$, ssi M est plus fréquent que M' dans $\mathcal{T}^-$. Si les deux motifs couvrent exactement le même ensemble de transactions dans $\mathcal{T}^-$, alors le motif le moins fréquent dans $\mathcal{T}^+$ sera préféré. En exploitant la relation de dominance $\succ_{\mathcal{R}}$, nous proposons la notion de top-k motif suspect.

Definition 5 (top-k motif suspect). *Un motif M est un top-k motif suspect (par rapport à $\succ_{\mathcal{R}}$) ssi $\nexists P_1, \dots P_k, \in \mathcal{L}_{\mathcal{I}}, \forall 1 \leq j \leq k,\ P_j \succ_{\mathcal{R}} M$.*

Ainsi, un motif M est un top-k motif suspect s'il n'existe pas plus de $(k-1)$ motifs qui le dominent. Un ensemble de top-k motifs suspects est défini comme suit :

$$\{M \in \mathcal{L}_{\mathcal{I}} \mid \texttt{élémentaire}(M) \wedge \nexists P_1, \dots P_k, \in \mathcal{L}_{\mathcal{I}}, \forall 1 \leq j \leq k,\ P_j \succ_{\mathcal{R}} M\}$$

La contrainte $\texttt{élémentaire}(M)$ permet de préciser que le motif M doit satisfaire une propriété de base. Dans notre cas, nous imposons que le motif recherché doit satisfaire les propriétés suivantes : $fermé_m(M) \wedge freq_{\mathcal{T}^-}(M) \geq 1 \wedge taille(M) \geq 1$.

4. Il s'agit d'une version décomposée de l'équation (2) sur $\mathcal{T}^+$ et $\mathcal{T}^-$.

4.3 Localisation de fautes par post-traitement des top-k motifs

Notre approche top-k retourne un ensemble ordonné de k meilleurs motifs $\langle TK_1, \ldots, TK_k \rangle$. Chaque motif TK_i représente un sous-ensemble d'instructions pouvant expliquer et localiser la faute. D'un motif à un autre, des instructions apparaissent/disparaissent en fonction de leurs fréquences dans les bases positives/négatives. Nous présentons dans cette section l'algorithme 1 qui prend en entrée les top-k motifs les plus suspects et retourne une liste S contenant un classement des instructions susceptibles de localiser la faute (ligne 12). La liste S inclut trois listes ordonnées (ID, IF et IA). Les éléments de ID sont classés en premier, suivis des éléments de IF et enfin des éléments de IA. Les listes ID et IA sont initialisées à vide, alors que la liste IF est initialisée avec les instructions qui apparaissent dans TK_1 (le plus suspect des motifs) (ligne 1). Ensuite, l'algorithme 1 essaye de différencier les instructions de IF en tirant profit des trois observations suivantes :

1. Comme TK_1 correspond au motif le plus suspect, il est susceptible de contenir l'instruction fautive, mais aussi d'autres instructions qui ont un certain degré de suspicion. C'est pourquoi, dans l'algorithme 1, IF est initialisée à TK_1 à la ligne 1.
2. Dans l'algorithme 1, les instructions qui sont dans IF et qui ne sont plus dans $TK_i (i = 2..k)$ sont notées par T_m (ligne 3). Ici, nous distinguons trois cas :
   (a) TK_i a la même fréquence que TK_1 dans la base négative mais le motif TK_i est plus fréquent dans la base positive que le motif TK_1. Ainsi, les instructions de T_m sont moins fréquentes dans la base positive comparée à celles de $(IF \setminus T_m)$. Ainsi, les instructions de T_m sont alors plus suspectes que les instructions restantes dans $IF \setminus T_m$ et doivent être classées en premier (supprimées de IF (ligne 6) et ajoutées à ID (ligne 5)).
   (b) TK_i est moins fréquent dans la base négative que TK_1. Encore une fois, les instructions de T_m doivent être classées en premier et ajoutées à ID.
   (c) Ainsi, ID contient les instructions les plus suspectes provenant de l'état initial de IF. Les instructions restantes dans IF qui apparaissent dans tous les motifs TK_i sont classées après celles de ID.
3. Certaines instructions T_p qui ne sont pas dans IF et qui apparaissent dans les autres motifs TK_i (ligne 7) doivent être ajoutées à IA et classées après IF (ligne 11).

En opérant ainsi, la première liste ID classe les instructions de TK_1 selon leur ordre de disparition dans $TK_2..TK_k$. La liste IF contient les instructions restantes de TK_1 qui apparaissent dans tous les motifs $TK_i (i = 1..k)$. Finalement, la liste IA contient les instructions qui n'appartiennent pas à TK_1 et qui apparaissent graduellement dans $TK_2..TK_k$.

Algorithm 1: Localisation de fautes par les top-k motifs

Input top-k motifs $TK = (TK_1, \ldots, TK_k)$
Output Liste ordonnée des instructions suspectes S

$ID \leftarrow \langle\rangle$; $IA \leftarrow \langle\rangle$; $IF \leftarrow TK_1$; $S \leftarrow \langle\rangle$;
foreach $i \in [2, k]$ **do**
  $T_m \leftarrow IF \setminus TK_i$;
  if $T_m \neq \emptyset$ **then**
    $ID.add(T_m)$;
    $IF.removeAll(T_m)$;
  end
  $Tp \leftarrow TK_i \setminus TK_{i-1}$;
  ia$\leftarrow \emptyset$;
  foreach $b \in Tp$ **do**
    if $(\forall$ia' $\in IA, \forall$id' $\in ID : b \notin$ia' $\wedge b \notin$id' $)$ **then** ia $\leftarrow$ ia $\cup\{b\}$;
  end
  if ia $\neq \emptyset$ **then** $IA.add($ia$)$;;
end
$S.addAll(ID)$; $S.add(IF)$; $S.addAll(IA)$;
return S ;

5 Expérimentations

Nous avons réalisé une série d'expérimentations en se comparant avec l'approche Tarantula [10] qui est un outil de référence en localisation de fautes. Notre mise en œuvre informatique, nommée F-CPminer, implémente notre approche avec un modèle CSP pour l'extraction des top-k motifs en utilisant Gecode [5], un solveur ouvert, libre et portable, pour le développement des programmes à contraintes. Nous avons aussi implémenté dans F-CPminer l'algorithme 1 détaillé dans la section précédente. Nos expérimentations ont été réalisées avec un processeur Intel Core i5-2400 2.40 GHz et 8 GO de RAM. Nous considérons la base des programmes Siemens [9] qui sont parmi les plus utilisés dans les études comparatives entre les techniques de localisation. Cette base est composée de sept programmes écrits en C, où chaque programme se décline en plusieurs versions ayant différentes fautes. Notons que des suites de cas de test sont disponibles pour tester chaque catégorie de programmes [6]. Nous avons exclu 21 versions qui sont hors de portée de la tâche de localisation de fautes (e.g., erreurs de segmentation). En somme, nous avons 111 programmes avec des fautes, détaillés dans la table 1.

Dans un premier temps, nous repérons les instructions couvertes par l'exécution d'un cas de test donné en faisant appel à l'outil Gcov [7]. Dans notre approche, nous avons besoin seulement de la matrice de couverture dont la gé-

5. www.gecode.org
6. une description complète de Siemens suite est donnée dans [9]
7. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

TABLE 1: La base Siemens (111 programmes)

Programme	Description	Versions erronées	LOC	Cas de tests
PrintTokens	Lexical analyzer	4	472	4071
PrintTokens2	Lexical analyzer	9	399	4056
Replace	Pattern replacement	29	512	5542
Schedule	Priority scheduler	5	292	2650
Schedule2	Priority scheduler	8	301	2680
Tcas	Altitude separation	37	141	1578
TotInfo	Information measure	19	440	1052

nération nécessite l'exécution du programme P sur chaque cas de test, puis le résultat retourné est comparé avec le résultat attendu. Si les deux résultats sont égaux, nous ajoutons la couverture du cas de test à la base transactionnelle positive ; autrement la couverture est ajoutée à la base négative.

Nous avons implémenté les formules statistiques de TARANTULA en calculant la suspicion des instructions de la même manière que celle présentée dans [10]. En particulier, nous avons adopté une métrique très connue dans la localisation de fautes, à savoir l'$\mathcal{EXAM}$ score [17] qui mesure l'efficacité d'une approche donnée de localisation. L'$\mathcal{EXAM}$ score donne le pourcentage des instructions qu'un développeur doit vérifier avant d'atteindre celle contenant la faute. Il est donc logique que la meilleure méthode est celle qui a le pourcentage d'$\mathcal{EXAM}$ score le plus petit.

TARANTULA et F-CPMINER peuvent retourner un ensemble d'instructions équivalentes en terme de suspicion (i.e., avec le même degré de suspicion). Dans ce cas, l'efficacité dépend de l'instruction qui est examinée en premier. Pour cette raison, nous donnons deux $\mathcal{EXAM}$ scores, l'$\mathcal{EXAM}$ score optimiste (resp. pessimiste) lorsque la première (resp. la dernière) instruction à examiner dans l'ensemble des instructions suspectes, est l'instruction contenant la faute.

TABLE 2: Comparaison par paires entre TARANTULA et F-CPMINER

Winner	Print	Print2	Replace	Sched	Sched2	Tcas	Totinfo	**Total**
F-CPMINER	**(2,2)**	**(6,7)**	(13,13)	(1,1)	**(7,7)**	**(19,20)**	**(14,14)**	**(62,64)**
TARANTULA	(0,0)	(0,0)	**(15,14)**	**(4,4)**	(1,1)	(6,4)	(2,0)	**(28,23)**
TIE GAME	(2,2)	(3,2)	(1,2)	(0,0)	(0,0)	(12,13)	(3,5)	**(21,24)**

(pessimiste, optimiste)

La table 2 donne une comparaison par paires entre TARANTULA et F-CPMINER sur les 111 programmes. Au niveau de chaque classe de programmes, nous donnons le nombre de versions où TARANTULA ou F-CPMINER gagne en comparant les valeurs de l'$\mathcal{EXAM}$ score retournées. Nous donnons aussi le cas où les valeurs de l'$\mathcal{EXAM}$ score sont égaux au niveau des deux méthodes (Tie game). Par exemple, prenons la classe Tcas ; avec l'$\mathcal{EXAM}$ score pessimiste (resp. optimiste), notre outil F-CPMINER a une meilleure efficacité sur 19 (resp. 20) versions, tant dis que TARANTULA est meilleur sur seulement 6 (resp. 4) versions.

Notons également que sur 12 (resp. 13) versions les deux approches sont équivalentes. La table 2 montre que F-CPMINER est très concurrentiel par rapport à TARANTULA en rapport avec l'efficacité (soit en pessimiste ou en optimiste). Cela est particulièrement vrai sur 5 classes de programmes sur 7 (voir les résultats en gras). F-CPMINER gagne au total sur 60% des programmes considérés dans le benchmark (62 à 64 sur 111 programmes). Cependant, TARANTULA gagne sur seulement 25% (23 à 28 sur 111 programmes) et donne le même résultat que F-CPMINER sur 15% des programmes (21 à 24 sur 111 programmes).

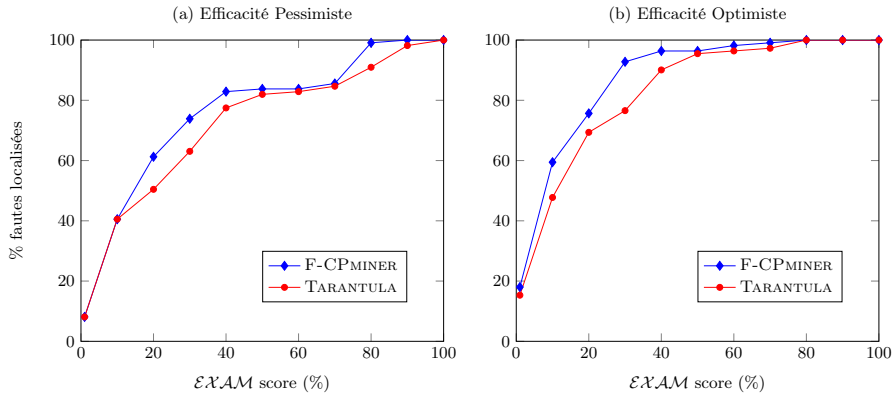

FIGURE 2: Comparaison de l'efficacité de TARANTULA et de F-CPMINER

Les figures 2a et 2b montrent une comparaison sur l'efficacité en termes d'$\mathcal{EXAM}$ score (pessimiste et optimiste) entre TARANTULA et F-CPMINER sur l'ensemble des programmes de Siemens. L'axe des x donne les valeurs de l'$\mathcal{EXAM}$ score et l'axe des y donne le pourcentage cumulatif des fautes localisées. Commençons par la figure 2a et l'$\mathcal{EXAM}$ score pessimiste. Jusqu'à une valeur de 10%, les deux outils agissent de la même manière en localisant plus de 40% des fautes. De 10% à 20%, nous observons une différence de 10% de fautes localisées par F-CPMINER comparé à TARANTULA (i.e, 60% pour F-CPMINER au lieu de 50% des fautes localisées pour TARANTULA). Par la suite, cette différence est réduite à 5% dans un intervalle de 20% à 40% de l'$\mathcal{EXAM}$ score. Après 70% de l'$\mathcal{EXAM}$ score, nous observons que TARANTULA a rattrapé F-CPMINER en localisant 80% des fautes. Les 20% des fautes restantes sont rapidement localisées par F-CPMINER (de 70% à 80% du $\mathcal{EXAM}$ score) alors que TARANTULA continue jusqu'à la fin pour localiser toutes les fautes (jusqu'à 100% du $\mathcal{EXAM}$ score).

Pour l'$\mathcal{EXAM}$ score optimiste illustré dans la figure 2b, F-CPMINER agit assez rapidement dés le début en localisant plus de fautes que TARANTULA. Il est aussi important de noter que les deux courbes ne s'intersectent pas et que celle

de F-CPminer est toujours au dessus de celle de Tarantula. D'une manière générale, nous pouvons conclure que F-CPminer est capable de localiser la plupart des fautes plus rapidement que Tarantula en terme d'efficacité (i.e., $\mathcal{EXAM}$ score). Notons que le coût en temps de la deuxième étape ad-hoc est négligeable par rapport à la première étape de recherche des top-k motifs.

Nous concluons cette section avec un dernier résultat intéressant en illustrant l'$\mathcal{EXAM}$ score cumulatif sur les 111 programmes. Nous donnons le nombre total d'instructions qui doivent être examinées par les deux méthodes sur l'ensemble des 111 programmes. F-CPminer doit examiner 2871 instructions contre 3432 pour Tarantula dans le cas du $\mathcal{EXAM}$ score pessimiste. Dans le cas optimiste, F-CPminer doit examiner 1822 instructions contre 2432 pour Tarantula.

6 Conclusion

Dans ce papier nous avons présenté une nouvelle approche basée sur la fouille de motifs ensemblistes et la programmation par contraintes (PPC) pour traiter le problème de localisation de fautes. Notre approche procède en deux étapes. En premier, nous formalisons le problème de localisation de fautes comme une problématique de fouille utilisant la PPC pour modéliser et résoudre les contraintes tenant compte de la nature bien particulière des motifs recherchés. La résolution du modèle à contraintes nous permet d'obtenir les top-k suites des instructions appartenant le plus souvent aux tests négatifs et le moins souvent aux test positifs. Nous obtenons ainsi les instructions les plus suspectes. La seconde étape ad-hoc a pour objectif de classer de façon plus précise l'ensemble des instructions des top-k en exploitant plusieurs observations sur la forme des motifs trouvés. Finalement, nous avons comparé expérimentalement notre approche implémentée dans l'outil F-CPminer et l'approche standard Tarantula sur la base Siemens.

Références

1. Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw., Pract. Exper.*, 23(6) :589–616, 1993.
2. Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Dellis : A data mining process for fault localization. In *Proceedings of SEKE'2009*, pages 432–437, 2009.
3. Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *Proceedings of SEKE'2011*, pages 238–243, 2011.
4. B. Crémilleux and A. Soulet. Discovering knowledge from local patterns with global constraints. In *ICCSA (2)*, pages 1242–1257, 2008.
5. L. De Raedt and A. Zimmermann. Constraint-based pattern set mining. In *Proceedings of the Seventh SIAM International Conference on DM*. SIAM, 2007.
6. Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD*, pages 204–212. ACM, 2008.

7. W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2) :188–208, 2010.
8. Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining : A constraint programming perspective. *Artificial Intelligence*, 175(12) :1951–1983, 2011.
9. Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th ICSE*, pages 191–200, 1994.
10. James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of IEEE/ACM ASE'05*, pages 273–282. ACM, 2005.
11. James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.
12. M. Khiari, P. Boizumault, and B. Crémilleux. Constraint programming for mining n-ary patterns. In *CP'10*, volume 6308 of *LNCS*, pages 552–567. Springer, 2010.
13. J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability : Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
14. Syeda Nessa, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *Wireless Algorithms, Systems, and Applications*, pages 548–559. Springer, 2008.
15. Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE, 2003.
16. W. Ugarte, P. Boizumault, S. Loudni, B. Crémilleux, and A. Lepailleur. Mining (soft-) skypatterns using dynamic CSP. In *CPAIOR*, pages 71–87, 2014.
17. W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.