Multiple Fault Localization Using Constraint Programming and Pattern Mining

Noureddine Aribi, Mehdi Maamar, Nadjib Lazaar, Yahia Lebbah, Samir Loudni

Multiple fault localization using constraint programming and pattern mining

Noureddine Aribi*, Mehdi Maamar†, Nadjib Lazaar‡, Yahia Lebbah*, Samir Loudni§
*LITIO – University of Oran 1, 31000 Oran – Algeria
†CNRS - CRIL, University of Artois, 62307 Lens – France
‡LIRMM – University of Montpellier, 34090 Montpellier – France
§CNRS, UMR 6072 GREYC – University of Caen Normandy, 14032 Caen – France

***Abstract*—Fault localization problem is one of the most difficult processes in software debugging. The current constraint-based approaches draw strength from declarative data mining and allow to consider the dependencies between statements with the notion of *patterns*. Tackling large faulty programs is clearly a challenging issue for Constraint Programming (CP) approaches. Programs with multiple faults raise numerous issues due to complex dependencies between faults, making the localization quite complex for all of the current localization approaches. In this paper, we provide a new CP model with a global constraint to speed-up the resolution and we improve the localization to be able to tackle multiple faults. Finally, we give an experimental evaluation that shows that our approach improves on CP and standard approaches.**

I. Introduction

Developing software programs is universally acknowledged as an error-prone task. The major bottleneck in software debugging is how to identify where the bugs are [26], this is known as fault localization problem. Nonetheless, locating a fault is still an extremely time-consuming and tedious task. Over the last decade, several automated techniques have been proposed to tackle this problem.

Spectrum-based approaches. *Spectrum-based fault localization* (SBFL) (e.g. [1], [18]) is a class of popular fault localization approaches that take as input a set of failing and passing test case executions, and then highlight the suspicious program statements that are likely responsible for the failures. A *ranking metric* is used to compute a suspiciousness score for each program statement based on observations of passing and failing test case execution. The basic assumption is that statements with high scores, i.e. those executed more often by failed test cases but never or rarely by passing test cases, are more likely to be faulty. Several ranking metrics have been proposed to capture the notion of suspiciousness, such as Tarantula [17], Ochiai [2], and Jaccard [2].

Multiple fault programs. Most of current localization techniques are based on the *single fault hypothesis*. By dismissing such assumption, faults can be tightly dependent in a program, giving rise to numerous behaviours [11]. Thus, it makes the localization process difficult for multiple fault approaches [3], [23], [16]. The main idea of these approaches is to make a partition on test cases into fault-clusters where each one contains the test cases covering the same fault. The drawback is that a test case can cover many faults with overlapping clusters, which leads to a rough localization. Another idea consists in localizing one fault at a time [27]. Here, we start by locating a first fault, then correct it (which is an error-prone step), generate again new test cases, and so on until no fault remains in the program.

Data Mining and CP based approaches. In the last decade, fault localization was abstracted as a data mining (DM) problem. Cellier et al. [6], [7] propose a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization. Maamar et al. [21] formalize the problem of fault localization as a closed pattern mining problem. A CP model with reified constraints is used to extract the k best patterns satisfying a set of constraints modeling the most suspicious statements. The drawback is the wide use of reified constraints, which causes problem scaling. Moreover, it considers only simple faults. Other approaches tackle fault localization as a supervised learning problem [4], [24]. CP-based approaches have also been investigated. Bekkouche et al. [5] proposed LocFaults a new tool for fault localization in a program for which a counter-example is available.

Declarative Data Mining. The CP-paradigm is at the core of generic approaches for pattern mining [9], [14]. With the recent development of global constraints, CP became competitive for solving some data-mining tasks [19], [20]. Recently, a large effort was made by the community to a better understanding of the unstructured information conveyed by the patterns and to produce *pattern sets* [10]. Mining top-k patterns (i.e. the k best patterns according to a score function) is a promising road to discover useful pattern sets.

The contribution of this paper is two-fold. Firstly, we propose a new CP model linked to multiple fault context for mining top-k suspicious patterns using a global constraint and a new scoring function based on the notion of *pattern suspiciousness degree* while ensuring the coverage criterion. Secondly, we give a new multiple fault ranking algorithm reasoning on the extracted top-k patterns to sort the statements from the most suspicious to the guiltless one. This algorithm exploits some observations linked to multiple fault localization and properties related to DM for finer-grained localization.

Experiments performed on single and multiple faults programs coming from Siemens Suite benchmarks show that our approach enables to propose a more precise localization as compared to the popular spectrum-based fault-localization approaches and the CP based approach F-CPMINER.

II. BACKGROUND

A. Fault Localization

Let us consider a faulty program `Prog` having n lines, labeled e_1 to e_n. For instance, we have $\texttt{Prog} = \{e_1, e_2, \ldots, e_{10}\}$ for the *Character Counter* program given in Fig.1. A **test case** tc_i is a tuple $\langle D_i, O_i \rangle$, where D_i is the input data and O_i is the expected output. Let $\langle D_i, O_i \rangle$ a test case and A_i be the current output returned by a program `Prog` after the execution of its input D_i. If $A_i = O_i$, tc_i is considered as a *passing* (i.e. positive), *failing* (i.e. negative) otherwise. A **test suite** $T = \{tc_1, tc_2, ..., tc_m\}$ is a set of m test cases to check whether the program `Prog` follows a given set of requirements.

Given a test case tc_i and a program `Prog`, the set of executed (at least once) statements of `Prog` with tc_i is called a *test case coverage* $I_i = (I_{i,1}, ..., I_{i,n})$, where $I_{i,j} = 1$ if the j^{th} statement is executed, 0 otherwise. I_i indicates which parts of the program are active during a specific execution. For instance, the test case tc_4 in Fig.1 covers the statements $\langle e_1, e_2, e_3, e_4, e_6, e_7, e_{10} \rangle$. The corresponding test case coverage is then $I_4 = (1, 1, 1, 1, 0, 1, 1, 0, 0, 1)$.

SBFL techniques assign suspiciousness scores for each of statements and rank them in a descending order of suspiciousness. Most of suspiciousness metrics are defined manually and analytically on the basis of multiple assumptions on programs, test cases and the introduced faults. Fig 2 lists the formula of three well-known metrics: TARANTULA [17], OCHIAI [2] and JACCARD [2]. Given a statement e_i, $pass(T)$ (resp. $fail(T)$) is the set of all passed (resp. all failed) test cases. $pass(e_i)$ (resp. $fail(e_i)$) is the set of passed (resp. failed) test cases covering e_i. The basic assumption is that the program fails when the faulty statement is executed. Moreover, the whole of suspiciousness metrics shares the same intuition: the more often a statement is executed by failing test cases, and the less often it is executed by passing test cases, the more suspicious the statement is considered. Fig.2 shows the suspiciousness spectrum of the different metrics according to an up to 1000 passing and/or failing test cases.

- TARANTULA allows some tolerance for the fault to be executed by passing test cases (see Fig.2a). However, this metric is not able to differentiate between statements that are not executed by passing tests. For instance, consider two statements e_i and e_j with $pass(e_i) = pass(e_j) = \emptyset$, $|fail(e_i)| = 1$ and $|fail(e_j)| = 1000$. For TARANTULA, e_i and e_j have the same suspiciousness degree.
- OCHIAI came originally from molecular biology. The specificity of this metric is that it attaches a particular importance of the presence to a statement in the failing test cases (see Fig.2b).
- JACCARD has been defined to find a proper balance between the impact of passing/failing test cases on the scoring measure [2] (see Fig.2c).

B. Closed Frequent Pattern Mining (CFPM)

Let $\mathcal{I} = \{1, ..., n\}$ be a set of n *items* indices and $\mathcal{T} = \{1, ..., m\}$ a set of *transactions* indices. A pattern P (i.e., itemset) is a subset of $\mathcal{I}$. The language of patterns corresponds to $\mathcal{L}_{\mathcal{I}} = 2^{\mathcal{I}}$. A transaction database is a set $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$. The set of items corresponding to a transaction identified by t is denoted $\mathcal{D}[t] = \{i \,|\, (i, t) \in \mathcal{D}\}$. A transaction t is an occurrence of some pattern P iff the set $\mathcal{D}[t]$ contains P (i.e. $P \subseteq \mathcal{D}[t]$). The cover of a pattern P is the set of all identifiers of transactions in which P occurs: $cover_{\mathcal{D}}(P) = \{t \in \mathcal{T} \,|\, P \subseteq \mathcal{D}[t]\}$. The frequency of a pattern P is the size of its cover: $freq_{\mathcal{D}}(P) = |cover_{\mathcal{D}}(P)|$.

Given a user-specified *minimum support* $\theta \in \mathbb{N}^+$, a pattern P is called *frequent* in $\mathcal{D}$ iff $freq_{\mathcal{D}}(P) \geq \theta$. The goal of frequent pattern mining is to identify all patterns $P \in \mathcal{L}_{\mathcal{I}}$ that are frequent in a given transaction database $\mathcal{D}$.

In mining frequent patterns, the main observation is that the output is often huge, particularly when the transaction database is dense, or using a too low minimum support threshold. As a consequence, several proposals have been made to generate only a *concise representation* of all frequent patterns. The most popular one is the so called *closed frequent patterns* [22].

Definition 1 (Closed frequent pattern): A frequent pattern $P \in \mathcal{L}_{\mathcal{I}}$ is closed if there does not exist a superset that has the same frequency: $closed_{freq}(P) \Leftrightarrow freq_{\mathcal{D}}(P) \geq \theta \,\wedge\, \forall i \in \mathcal{I} \setminus P : freq_{\mathcal{D}}(P \cup \{i\}) < freq_{\mathcal{D}}(P)$.

The user is often interested in discovering richer patterns satisfying properties defined on a set of patterns [10]. In this setting, the approach that we present in this paper deals with top-k patterns, which are the k best patterns optimizing an interestingness measure.

Definition 2 (top-k patterns): Let m be a measure, k an integer and $\rhd$ a dominance relation on $\mathcal{L}_{\mathcal{I}}$. top-$k$ is the set of k best patterns according to m : $\{x \in \mathcal{L}_{\mathcal{I}} \mid \; freq_{\mathcal{D}}(x) \geq 1 \,\wedge\, \nexists y_1, \ldots, y_k \in \mathcal{L}_{\mathcal{I}} : \; \forall 1 \leq j \leq k, m(y_j) \rhd m(x)\}$.

C. CP models for the itemset mining

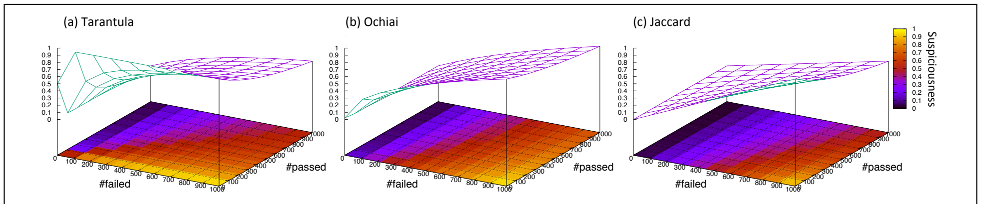
Two CP models have been proposed for mining closed frequent patterns. The first one is based on reified constraints [9], while the second one uses a global constraint [20] to express the mining task.

Reified model. De Raedt *et al.* have proposed in [9] a first CP model for itemset mining (`CP4IM`). They showed how some constraints (e.g. frequency, closedness) can be formulated using a CP approach [9], [13]. This modeling uses two sets of boolean variables: (1) item variables $\{P_1, P_2, ..., P_n\}$ where $(P_i = 1)$ iff $(i \in P)$; (2) transaction variables $\{T_1, T_2, ..., T_m\}$ where $(T_t = 1)$ iff $(P \subseteq t)$. The relationship between P and $\mathcal{T}$ is modeled by m reified n-ary constraints. The minimal frequency constraint is encoded by n reified m-ary constraints. The closedness constraint $closed_{freq}(P)$ is encoded by n m-ary constraints.

Global constraint model. Despite the declarative side of the reified model, such an encoding has a major drawback since it leads to constraints networks of huge size, by introducing extra variables and numerous constraints, precisely $(m{+}n{+}n)$

Program : Character counter	tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8	TARANTULA	OCHIAI	JACCARD	F-CPMINER	MULTILOC
	Test cases												
`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	8	4	4	4	7
e_2: `if('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	0	1	4	2	2	2	2
e_3: `let += 2; //fault1`	1	1	1	1	1	1	0	0	3	1	1	1	1
e_4: `else if ( 'a'<=c && 'z'>=c )`	1	1	1	1	1	0	0	1	5	5	5	5	10
e_5: `let += 1;`	1	1	0	0	1	0	0	0	3	7	7	7	4
e_6: `else if ( '0'<c && '9'>c ) //fault2`	1	1	1	1	0	0	0	1	6	6	6	6	3
e_7: `dig += 1;`	0	1	0	1	0	0	0	0	3	8	8	10	5
e_8: `else if (isprint (c))`	1	0	1	0	0	0	0	1	10	10	10	10	10
e_9: `other += 1;}`	1	0	1	0	0	0	0	1	10	10	10	10	10
e_{10}: `printf("%d %d %d\n", let, dig, other);}`	1	1	1	1	1	1	1	1	8	4	4	4	7
Passing/Failing	F	F	F	F	F	F	P	P					

Figure 1: "Character counter" program containing two faults.



$$T(e_i) = \frac{\frac{|fail(e_i)|}{|fail(T)|}}{\frac{|pass(e_i)|}{|pass(T)|} + \frac{|fail(e_i)|}{|fail(T)|}} \qquad O(e_i) = \frac{|fail(e_i)|}{\sqrt{(|fail(e_i)|+|pass(e_i)|) \times |fail(T)|}} \qquad J(e_i) = \frac{|fail(e_i)|}{|pass(e_i)|+|fail(T)|}$$

Figure 2: Suspiciousness Degrees.

constraints. Moreover, it does not ensure domain consistency [1] [20]. Space complexity and non-domain consistency of the reified approach are clearly identified as the two main bottlenecks behind the competitiveness of this declarative model. To address these two issues, the CLOSEDPATTERN global constraint was proposed [20], enabling to encode efficiently both the minimal frequency constraint and the closedness constraint. This global constraint does not require any reified constraint nor extra variables.

*Definition 3 (*CLOSEDPATTERN *global constraint):* The global constraint $\text{CLOSEDPATTERN}_{\mathcal{D},\theta}(P)$ holds iff there exists an assignment $\sigma = \langle d_1, ..., d_n\rangle$ of variables P s.t. $freq_{\mathcal{D}}(\sigma) \geq \theta$ and $closed_{freq}(\sigma)$.

Lazaar et al. [20] have proposed a filtering algorithm that ensures a domain consistency for the global constraint in worst case time $O(n^2 \times m)$. It ensures also extracting the whole set of closed patterns, with a backtrack free manner, having the worst case time $O(C \times n^2 \times m)$, where C is the number of closed patterns. It is the first CP model ensuring a tight complexity, very close to specialized data mining approaches, such as LCM [25].

III. MULTILOC APPROACH

This section presents our approach implemented in MULTILOC tool for multiple fault localization using CLOSEDPATTERN. It consists of two steps:

1) *top-k Extraction:* this step extracts the k most suspicious patterns. Unlike the approach proposed in [21], our top-k extraction exploits (i) a new measure (coined PSD for Pattern Suspiciousness Degree); (ii) a new CP model based on CLOSEDPATTERN global constraint [20]; (iii) the extracted top-k patterns ensure the coverage criterion. This criterion ensures that all statements are covered by the top-k patterns. Doing so, we are able to rank the overall statements.
2) *Ranking step.* The extracted top-k patterns are confronted each other to provide a meaningful localization. The ranking algorithm presented in [21] is based on observations motivated by simple faults. We propose a thorough analysis for a multiple fault ranking algorithm based on observations linked to multiple fault context as well as properties associated to pattern mining for a finer-grained localization.

Let $\texttt{Prog} = \{e_1, ..., e_n\}$ be a set of indexed statements composing the program Prog and $\mathcal{T} = \{tc_1, ..., tc_m\}$ a set of test cases. The transactional dataset $\mathcal{D}$ is defined as follows: (i) each statement of Prog corresponds to an item in $\mathcal{I}$, (ii) the coverage of each test case tc_i forms a transaction in $\mathcal{T}$. Moreover, to look for contrasts between subsets of transactions, $\mathcal{T}$ is partitioned into two disjoint subsets $\mathcal{T}^+$ and $\mathcal{T}^-$. $\mathcal{T}^+$ (resp. $\mathcal{T}^-$) denotes the set of coverage of positive (resp. negative) test cases.

Let d be the 0/1 (m, n) matrix representing the dataset $\mathcal{D}$. So, $\forall t \in \mathcal{T}, \forall i \in \mathcal{I}$, $(d_{t,i} = 1)$ if and only if the statement i is executed (at least once) by the test case t. Figure 1 shows the transactional dataset associated to the program *Character Counter*. For instance, the coverage of the test case t_5 is $I_5 = (1, 1, 1, 1, 1, 0, 0, 0, 0, 1)$. As t_5 fails, thus $I_5 \in \mathcal{T}^-$.

We propose a new measure that evaluates the suspiciousness degree of a pattern (i.e., subset of statements present in the same failing/passing test cases) rather than the suspiciousness of an isolated statement.

*Definition 4 (Pattern Suspiciousness Degree (*PSD*)):* Given

[1]Domain consistency, also known as arc consistency, enables to remove from the domain of each variable all values that do not belong to a solution of the considered constraint.

a pattern P of a program. The Suspiciousness degree of P is:

$$\text{PSD}(P) = freq^-(P) + \frac{|\mathcal{T}^+| - freq^+(P)}{|\mathcal{T}^+| + 1}$$

Where $freq^-(P)$ (resp. $freq^+(P)$) represents the frequency of the pattern P in the negative dataset $\mathcal{T}^-$ (resp. positive dataset $\mathcal{T}^+$).

The originality of PSD formula comparing to SBFL metrics comes, first and foremost, from the fact that PSD takes into account the dependencies between statements. Second, PSD attaches a particular importance to the presence of a pattern in the failing test cases. That is, with its unbounded part on failed tests, PSD is able to differentiate between two patterns P_1 and P_2 where $freq^-(P_1) > freq^-(P_2)$ and $freq^+(P_1) = freq^+(P_2)$. The more frequent pattern in failing test cases is more suspect than the less frequent, whatever their presence in passing ones. Now, to evaluate the interest of pattern in terms of suspiciousness w.r.t. a set of patterns, we define a dominance relation between patterns.

Definition 5 (PSD-dominance relation): A pattern P dominates another pattern P' w.r.t. PSD (denoted by $P \rhd_{\text{PSD}} P'$), iff : $\text{PSD}(P) > \text{PSD}(P')$

The PSD value of a given pattern is proportional to its presence in failing test cases ($freq^-$) and inversely related to its presence in passing test cases ($freq^+$). Thus, the dominance relation states that $P \rhd_{\text{PSD}} P'$, if P is more suspect than P'. The top-k suspicious patterns are extracted according to the definition 2 with the use of $\rhd_{\text{PSD}}$ as dominance relation. Thus, P is a top-k suspicious pattern if there exists no more than $(k-1)$ patterns that PSD-dominate P.

A. top-k Extraction

Algorithm 1 details the extraction of the top-k suspicious patterns ensuring the coverage criterion. It takes as input the faulty program `Prog`, the corresponding datasets of negative and positive test cases ($\mathcal{T}^-$ and $\mathcal{T}^+$). It returns as output top-k suspicious patterns covering all statements of the program `Prog`. The algorithm starts by building the following CP model (line 4):

- $X = \{X_1, .., X_n\}$ the binary statement variables where $(X_i = 1)$ if the statement e_i is in the searched pattern P
- C_α, the set of constraints composed of :
  - ClosedPattern$_{\mathcal{T},\theta}(X)$ with $(\theta = 1)$.
  - $\text{PSD}(X) > \alpha$ to ensure that the PSD value of the extracted pattern P must be greater than α.

This CP model aims at extracting a closed pattern in $\mathcal{T}$ dataset with a PSD greater than α. Starting with $\alpha = 0$, we compute the k first suspicious patterns (lines 5-8). During the search, a top-k list of suspicious patterns, noted $\mathcal{S}$, is maintained. Once the k patterns are found, $\mathcal{S}$ is sorted according to decreasing order $\rhd_{\text{PSD}}$ (line 9). It is worth noting that the PSD value of each pattern P in top-k list is calculated using the frequency of P in $\mathcal{T}^-$ and $\mathcal{T}^+$(i.e., $freq^-$ and $freq^+$). Considering that the first extracted patterns are not necessarily the top-k ones, we try iteratively to extract a pattern that PSD-dominates the last one S_k (lines 10-16). This is achieved by:

Algorithm 1: Extracting the top-k suspicious patterns.

1 **Input**: `Prog`, $\mathcal{T} = \{\mathcal{T}^-, \mathcal{T}^+\}$
2 **Output**: top-k suspicious patterns $\mathcal{S}$
3 $Loc \leftarrow \langle\rangle$; $k \leftarrow |$`Prog`$|$; $covered \leftarrow \emptyset$; $\alpha \leftarrow 0$; $i \leftarrow 1$;
4 $(X, C_\alpha) \leftarrow CPmodel(\mathcal{T})$;
5 **repeat**
6     $P \leftarrow SolveNext(X, C_\alpha)$;
7   **if** $P \neq \emptyset$ **then** $\mathcal{S}.add(P)$; $i++$;
8 **until** $(i > k) \vee (P = \emptyset)$;
9 Sort $\mathcal{S}$ according to decreasing order $\rhd$;
10 **repeat**
11     $\alpha \leftarrow \text{PSD}(S_k)$;
12     $P \leftarrow SolveNext(X, C_\alpha)$;
13   **if** $P \neq \emptyset$ **then**
14         $\mathcal{S}.remove(S_k)$;
15   Insert P in $\mathcal{S}$ according to decreasing order $\rhd_{\text{PSD}}$;
16 **until** $P = \emptyset$;
17 $covered \leftarrow \bigcup_{i \in 1..k} S_i$;
18 **if** $|covered| < k$ **then**
19     $\mathcal{R} \leftarrow$`Prog` $\setminus covered$;
20     $\alpha \leftarrow 0$;
21   **foreach** $e_i \in \mathcal{R}$ **do**
22         $C_\alpha.add(X_i = 1)$;
23         $P \leftarrow SolveNext(X, C_\alpha)$;
24         $C_\alpha.remove(X_i = 1)$;
25   Insert P in $\mathcal{S}$ according to decreasing order $\rhd_{\text{PSD}}$;
26 **return** $\mathcal{S}$;

(i) updating α with the PSD of S_k (line 11), (ii) removing the last pattern S_k (line 14), (iii) inserting the new P in the right place according to $\rhd_{\text{PSD}}$ order (line 15). This process is repeated until no pattern, better than the last S_k in terms of $\rhd_{\text{PSD}}$ can be found (line 16).

The lines 19-25 ensure the coverage criterion. If some statements $\mathcal{R}$ are not covered by any S_i, than for each statement in $\mathcal{R}$, we extract a pattern covering it. This is achieved by: (i) reinitializing α to 0, to be able to generate any pattern covering $e_i \in \mathcal{R}$ (line 20), (ii) adding the constraint $(X_i = 1)$ to the store C_α before extraction (line 22), (iii) removing $(X_i = 1)$ from the store C_α after extraction (line 24), (iv) inserting the new P in the right place according to $\rhd_{\text{PSD}}$ order (line 25). It is important to stress that the extraction at line 23 will always return a pattern. This follows from the assumption that each statement e_i in `Prog` is covered by at least a test case, and when the thresholds $(\alpha, \theta) = (0, 1)$.

B. The ranking Process

The top-k patterns represent a rough localization. The ranking step in MultiLoc approach confronts the top-k patterns to produce a finer-grained multiple fault localization.

Definition 6 (Highly suspect statements): Given two top-k patterns $S_i, S_j \in \mathcal{S}$ s.t., $S_i \rhd_{\text{PSD}} S_j$. A statement $e \in S_i \setminus S_j$ is highly suspect if $freq^+(e) < freq^+(S_j)$.

The following proposition shows that highly suspect relation preserves the PSD-dominance relation.

Algorithm 2: Ranking Step.

1 **Input** k patterns $\mathcal{S} = \langle S_1, \dots, S_k \rangle$;
2 **Output** an ordered list of suspicious statements Loc
3 $suspect \leftarrow \langle\rangle$; $pending \leftarrow \langle\rangle$; $guiltless \leftarrow \langle\rangle$;
4 **foreach** $i \in \{1, \dots, k-1\}$ **do**
5    $guiltless.addAll(\bigcap_{l \in i..k} S_l \setminus suspect)$;
6   **foreach** $j \in \{i+1, \dots, k\}$ **do**
7      $\Delta \leftarrow S_i \setminus S_j$;
8   **foreach** $e \in \Delta$ **do**
9   **if** $freq^+(e) < freq^+(S_j)$ **then**
10          $suspect.add(e)$;
11          $\Delta.remove(e)$;
12      $pending.add(\Delta)$;
13 $guiltless.addAll(S_k \setminus suspect \cup pending)$;
14 $Loc.addAll(suspect)$;
15 $Loc.addAll(pending \setminus suspect)$;
16 $Loc.addAll(guiltless)$;
17 **return** Loc;

Proposition 1: Given two top-k patterns $S_i, S_j \in \mathcal{S}$ s.t., $S_i \rhd_{\text{PSD}} S_j$ and a statement e highly suspect, then $\text{PSD}(e) > \text{PSD}(S_j)$ and $freq^+(S_i) < freq^+(S_j)$.

Proof: Since that $S_i \rhd_{\text{PSD}} S_j$, we have $\text{PSD}(S_i) > \text{PSD}(S_j)$, and $freq^-(S_i) \geq freq^-(S_j)$ (Def.4). From the anti-monotonicity property of the frequency (i.e., $X \subseteq Y \Rightarrow freq(Y) \leq freq(X)$), $freq^-(e) \geq freq^-(S_i)$ $(\geq freq^-(S_j))$. Taking into account $freq^+(e) < freq^+(S_j)$, we have $\text{PSD}(e) > \text{PSD}(S_j)$, but the converse is not true. We have $freq^+(S_i) < freq^+(e)$ $(< freq^+(S_j))$, thus $freq^+(S_i) < freq^+(S_j)$. □

Definition 7 (Guiltless statements): Given a top-k patterns $\mathcal{S} = \langle S_1, \dots, S_k \rangle$. Statements shared by the top-k patterns $\bigcap_{i \in 1..k} S_i$ are guiltless statements.

The intuition behind definition 7 is that statements that are always executed by failing/passing test cases, initialisation statements for instance, are less suspicious.

Algorithm 2 takes as input the top-k patterns and returns a ranked list Loc of most accurate suspicious statements enabling to better locate multiple faults. The algorithm is based on definitions 6, 7 and proposition 1. The returned list Loc includes three computed ordered lists noted $suspect$, $pending$ and $guiltless$. Elements of $suspect$ list are ranked first (line 14), followed by those of $pending$ list (line 15), then by elements of $guiltless$ (line 16). As we tackle multiple faults, a given fault can appear in any S_i. For that, the main loop at line 6 aims to confront S_i with the rest of top-k patterns S_j with $i < j$. At line 5, we compute the guiltless statements according to Def.7. Lines 7-8 allow us to emerge the highly suspect statements according to Def.6 by filling the $suspect$ list. After this treatment, the remaining statements are neither $suspect$ nor $guiltless$. So, they are added to the $pending$ list (line 12). At the end, we accord a particular treatment to the last top-k pattern S_k by computing what remains as statement in S_k not designated as $suspect$ or $guiltless$ and not in the $pending$ list (line 13). Once done, we add respectively in the localization Loc the $suspect$ list (line 14), the $pending$ list (line 15) and the less suspicious statements in the $guiltless$ list (line 16).

Table I: $\mathcal{S}$: top-k suspicious patterns of example 1

top-k	$freq^+$	$freq^-$	PSD
$S_1 : (e_1, e_2, e_3, e_{10})$	0	6	6.66
$S_2 : (e_1, e_2, e_{10})$	1	6	6.33
$S_3 : (e_1, e_{10})$	2	6	6
$S_4 : (e_1, e_2, e_3, e_4, e_{10})$	0	5	5.66
$S_5 : (e_1, e_2, e_4, e_{10})$	1	5	5.33
$S_6 : (e_1, e_2, e_3, e_4, e_6, e_{10})$	0	4	4.66
$S_7 : (e_1, e_2, e_4, e_6, e_{10})$	1	4	4.33
$S_8 : (e_1, e_2, e_3, e_4, e_5, e_{10})$	0	3	3.66
$S_9 : (e_1, e_2, e_3, e_4, e_6, e_7, e_{10})$	0	2	2.66
$S_{10} : (e_1, e_2, e_3, e_4, e_6, e_8, e_9, e_{10})$	0	2	2.66

IV. Running Example

To illustrate our approach, we consider the *Character counter* program given in Fig.1. In this figure, we have eight test cases where tc_1 to tc_6 are passing test cases, and tc_7 and tc_8 are failing test cases. According to the provided test cases, we report the suspiciousness ranking given by TARANTULA, OCHIAI and JACCARD. We give also the ranking given by the CP approach F-CPMINER. In this example, two faults are introduced at e_3 and e_6, where the correct statements are respectively "`let+ = 1`" and "`else if('0' <= c && '9' >= c)`". We fix the size of the top-k patterns to the size of the program (i.e., $k = 10$).

Table I shows the top-k patterns generated using Algo.1 and their corresponding $freq^+$, $freq^-$ and PSD values. Using Algo.2 on the extracted top-k, the first fault in e_3 is ranked first. Here, e_3 is the only statement that is in S_1 and disappears in S_2 (i.e., $e_3 \in S_1 \setminus S_2$). This statement satisfies the proposition 1 with $freq^+(e_3) < freq^+(S_2)$ and thus, it is highly suspect and added to $suspect$ list in Algo.2. OCHIAI and JACCARD are also able to rank e_3 on the top while TARANTULA ranked it in the third position by missing the behaviour of such fault with its formulation. F-CPMINER is also able to rank e_3 on the top with its dedicated reasoning on single fault.

In the same way, e_2 is ranked in the second position with MULTILOC using proposition 1 on S_2 and S_3. Afterwards, no more statements are added to the $suspect$ list. Most of them are added to the $guiltless$ list like e_1 and e_{10} where $\bigcap_{i \in 1..10} S_i = \{e_1, e_{10}\}$. The localization Loc returned by Algo.2 is built first with $suspect$ list, and second with $pending$ list. Here we can observe that e_6 is the first statement added to $pending$ list and thus, ranked in the third position (Fig.1). If we take a look to S_6 and S_8, e_6 is disappearing but does not satisfy proposition 1 where $freq^+(e_6) > freq^+(S_8)$. Thus, e_6 is added to $pending$ list in Algo.2. To sum up, Algo.2 on top-k patterns given in Table I will build:

- $suspect = \langle\langle e_3\rangle, \langle e_2\rangle\rangle$
- $pending = \langle\langle e_6\rangle, \langle e_5\rangle, \langle e_7\rangle\rangle$
- $guiltless = \langle\langle e_1, e_{10}\rangle, \langle e_4\rangle, \langle e_8, e_9\rangle\rangle$
- $Loc = \langle\langle e_3\rangle, \langle e_2\rangle, \langle e_6\rangle, \langle e_5\rangle, \langle e_7\rangle, \langle e_1, e_{10}\rangle, \langle e_4\rangle, \langle e_8, e_9\rangle\rangle$

Table II: Siemens suite.

Program	# Versions	LOC	LEC	Test cases	$\lvert\mathcal{T}^+\rvert$	$\lvert\mathcal{T}^-\rvert$
Replace	29	514	245	5542	5450	92
PrintTokens2	9	358	200	4056	3827	229
PrintTokens	4	348	195	4071	4016	55
Schedule	5	294	152	2650	2506	144
Schedule2	8	265	128	2680	2646	34
TotInfo	19	272	123	1052	1015	37
Tcas	37	135	65	1578	1542	36

LOC: lines of code in the correct version – LEC: lines of executable code

Taking a look to the other approaches, SBFL metrics are ranking e_6 in the same position, the sixth one. A same ranking is obtained using F-CPMINER with its single fault reasoning.

V. EXPERIMENTS

In this section, we present the experimental results obtained on some benchmarks. First, we describe the benchmark programs. Second, we present the experimental protocol and our implementation. Third, we provide the results and comparisons with existing approaches.

A. Benchmark programs

We have considered the Siemens suite[2], which is the most common program suite used to evaluate software testing and fault localization approaches. The Siemens suite is provided with seven C programs, each program has a correct version and a set of faulty versions (one fault per version). The suite is also provided with test suites for each faulty version.

Single Fault benchs. Table II summarizes the 111 faulty programs used in our single fault experiments. For each Siemens program, we report the number of faulty versions, the size of the program with its lines of code (LOC) and lines of executable code (LEC), the averaged number of test cases, passing and failing test cases.

Multiple Faults benchs. Based on the Siemens suite and by combining randomly the provided faults, we create 6 versions with two faults, 6 versions with three faults and 3 versions with four faults. In addition, we ensured that each fault is in a different statement.

B. Experimental protocol

First, we need to know the statements that are covered by a given (passing/failing) test case. For this end, we use GCOV [3] profiler tool to trace and save the coverage information of test cases as a boolean matrix (e.g., see Fig.1). Then, each test case is classified as positive/negative w.r.t. the provided correct version. Doing so, we get two datasets $(\mathcal{T}^+, \mathcal{T}^-)$ for each faulty program version.

We have implemented our Multiple fault localization approach as a tool coined MULTILOC. The first step with the CP model and Algo.1 are implemented within GECODE[4] solver (where the CLOSEDPATTERN global constraint was implemented). The ranking step with Algo.2 is implemented in C++ and linked to the first step. Our experiments were conducted on an Intel® i5-2400 CPU at 3.10GHz x 4 with 8 GB RAM. The CPU timeout was set to $180s$, which is an acceptable waiting time for a localization.

For a fair comparison between our tool and the other approaches, we have implemented the SBFL metrics, TARANTULA, OCHIAI and JACCARD as presented in [17], [2]. We have also used the distribution of F-CPMINER provided by the authors [21].

The localization accuracy is evaluated with the notion of EXAM score [27]. The EXAM score measures the total effort given by a developer to locate the fault. In other words, it reports the percentage of statements that a developer will check before the one containing the fault: lower is better. This being said, some statements can be equivalent in terms of suspiciousness. Here, the accuracy may vary depending on which statement to check first. For such reason, we report two exam scores, the *optimistic* and the *pessimistic* one, denoted respectively O-EXAM and P-EXAM. We talk about O-EXAM (resp. P-EXAM) when the first (resp. last) statement to check in the set of equivalent statements is the faulty one.

[2]A complete description of Siemens suite can be found in [12], [15]. Siemens suite is available on http://sir.unl.edu/php/previewfiles.php

[3]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[4]www.gecode.org

Table III: Qualitative comparison for single fault on Siemens Suite (Exam score %). **(1):** MULTILOC **(2):** F-CPMINER **(3):** TARANTULA **(4):** OCHIAI **(5):** JACCARD

Program	P-EXAM (%)					O-EXAM (%)				
	(1)	(2)	(3)	(4)	(5)	(1)	(2)	(3)	(4)	(5)
Replace	**5.67**	12.34	11.09	8.07	10.81	**4.45**	10.47	9.35	6.34	9.07
PrintTokens2	**2.55**	2.66	16.52	10.83	16.07	**1.61**	1.72	15.58	9.88	15.13
PrintTokens	**3.97**	5.89	13.59	6.66	11.28	**2.05**	3.97	11.66	4.74	9.35
Schedule	17.72	25.26	6.70	**6.01**	6.30	16.01	22.63	4.99	**4.31**	4.60
Schedule 2	47.04	**44.82**	61.70	55.29	61.61	36.24	**30.46**	50.90	44.49	50.80
Tot info	11.81	**11.51**	23.62	18.05	21.52	6.07	**5.56**	17.88	12.32	15.78
Tcas	43.11	**40.33**	43.94	42.11	43.86	19.08	**16.13**	19.91	18.09	19.83
Total	**18.83**	20.40	25.45	21.17	24.62	**12.22**	12.99	18.72	14.44	17.89

C. Single Fault Results

Table III reports an EXAM score comparison (P-EXAM and O-EXAM) between MULTILOC, F-CPMINER and SBFL metrics on single fault programs. Each line reports the averaged P-EXAM/O-EXAM scores of the different versions of the corresponding Siemens program. The first observation that we can draw is that, except *Schedule* versions, MULTILOC is more accurate than SBFL metrics (P-EXAM and O-EXAM score). On the whole, our approach is able to locate single faults better than F-CPMINER. Now, if we take a close look we can observe that F-CPMINER is better on programs with size less than 130 LEC. On the other side, MULTILOC is more efficient than F-CPMINER on programs with size more than 190 LEC.

As F-CPMINER and MULTILOC start by generating top-k patterns from $\mathcal{T}$ dataset, we run the two CP models and extract top-k patterns. Table IV reports the results in terms of CPU time (in seconds) and the search space size with the number of nodes. The main observation is that our CP model with the use of a dedicated global constraint for closed pattern outperforms significantly the CP model used in F-CPMINER at all levels. In terms of CPU times, we can observe for *tcas* programs (37 versions out of 111) a speed-up factor of 8. Factors of 19 to 32 are noted for five programs (65 versions out of 111) and for *Print Token2* (9 versions out of 111), we have a factor of

Table IV: Comparative performance for extracting top-k patterns. **(1):** MULTILOC **(2):** F-CPMINER

Program	k	Time (s) (1)	Time (s) (2)	Speed-up (2)/(1)	#Nodes (1)	#Nodes (2)	Factor (2)/(1)
Replace	245	**5.24**	147.69	28	**1056**	2450119	2320
PrintTokens2	200	**2.69**	146.25	54	**574**	3588108	6251
PrintTokens	195	**3.20**	61.49	19	**524**	1235562	2357
Schedule	152	**1.84**	14.89	32	**2299**	210884	91
Schedule 2	128	**0.47**	12.66	26	**579**	113438	195
TotInfo	123	**0.10**	2.53	25	**270**	10445	38
Tcas	65	**0.02**	0.16	8	**38**	133	3

54. The same observation, with more amplified factors, can be draw on the number of explored nodes.

D. Multiple Fault Results

Table V reports an EXAM score comparison (P-EXAM and O-EXAM) between MULTILOC, F-CPMINER and SBFL metrics on the 15 multiple fault versions. For instance, if we take programs with three faults, we have reported the averaged P-EXAM and O-EXAM scores of the first located fault $f1$, the second fault $f2$ and the third one $f3$.

Let us start with the two fault programs (6 versions). Our first observation is that CP approaches are drastically more efficient than SBFL metrics in terms of P-EXAM and O-EXAM. In general, CP approaches reduce by half the EXAM score for $f1$ and by approximately 20% for $f2$. As F-CPMINER is designed for single faults, we observe a slight improvement comparing to our approach on $f1$ in terms of P-EXAM and O-EXAM. However, MULTILOC with its multiple fault reasoning is quite more efficient on $f2$ than F-CPMINER (20% on P-EXAM and 8% on O-EXAM).

For three-faults programs (6 versions), here also CP approaches are more accurate than the standard SBFL metrics. With three faults, dependencies between faults are more significant giving rise to numerous behaviours affecting the localization process. F-CPMINER becomes less effective even on the first fault $f1$. MULTILOC clearly demonstrates a stability and robustness on multiple faults. Comparing with F-CPMINER, MULTILOC wins with 2% on $f1$, 5% on $f2$ and 20% on $f3$ in terms of P-EXAM and O-EXAM.

For four-faults programs (3 versions), here the results support our observations on the comparison between F-CPMINER and MULTILOC. Furthermore, we observe that F-CPMINER is not able to catch $f4$ in any program version (EXAM score at 100%). It is important to stress that such case can not happen with MULTILOC, since our approach ensures the coverage criterion. Comparing now with SBFL metrics, we observe that in some cases, the standard metrics can be effective. For instance, JACCARD is able to quickly locate the first fault $f1$. This can be explained by the fact that each metric was defined to catch a particular fault-behaviour. In multiple fault context, combining faults can lead to a broad spectrum of behaviour and each time one of them is catched by a given metric. However, on the whole, MULTILOC remains the robust approach on four-fault programs with an improvement of more than 15% on P-EXAM and O-EXAM comparing with SBFL metrics.

Figure 3 shows a comparison on P-EXAM (fig.3a) and O-EXAM (fig.3b) between SBFL metrics, F-CPMINER and

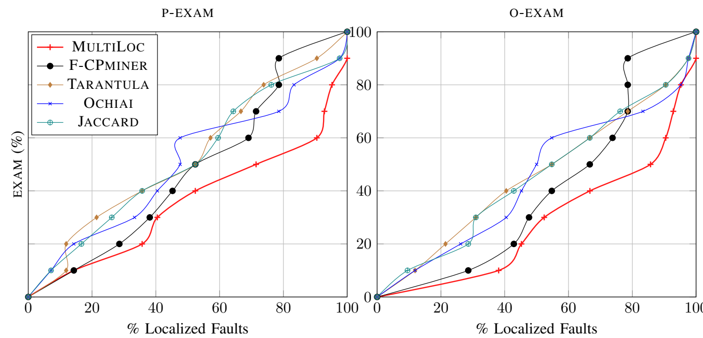


Figure 3: MULTILOC vs F-CPMINER vs `Measures` : Multiple Faults

Table V: Qualitative comparison for multiple faults on Siemens Suite (EXAM score %). **(1):** MULTILOC **(2):** F-CPMINER **(3):** TARANTULA **(4):** OCHIAI **(5):** JACCARD

Program		P-EXAM (%) (1)	(2)	(3)	(4)	(5)	O-EXAM (%) (1)	(2)	(3)	(4)	(5)
A	*f1*	15.36	**12.29**	33.75	26.35	30.93	8.59	**7.00**	26.84	19.44	24.02
	f2	**43.82**	62.70	68.87	62.79	66.97	**38.67**	46.75	65.80	59.72	63.90
B	*f1*	**13.01**	15.60	34.94	27.99	33.28	**7.53**	9.66	28.10	20.39	26.45
	f2	**39.96**	44.69	60.47	54.26	57.91	**26.31**	31.09	57.28	45.54	54.72
	f3	**49.84**	69.22	80.11	76.42	77.61	**40.96**	58.35	67.12	69.72	64.63
C	*f1*	21.77	22.83	8.77	10.30	**7.98**	21.46	22.42	8.50	7.68	**7.54**
	f2	39.00	56.28	**20.99**	41.26	23.99	35.72	51.09	**18.64**	40.99	21.81
	f3	58.71	83.61	**53.86**	59.46	66.78	**41.09**	81.28	53.58	59.18	66.51
	f4	**63.69**	100	78.01	75.17	75.17	**46.61**	100	76.23	73.39	73.40
Total		**37.80**	51.91	48.86	48.19	48.96	**29.66**	45.29	44.68	44.01	44.77

A: 2 faults B: 3 faults C: 4 faults

MULTILOC. The x-axis reports the cumulative percentages of located faults on the 15 multiple fault programs (42 faults in total) while y-axis reports the EXAM score. Let us start with the pessimistic case (i.e. P-EXAM). 15% of faults are located at the same time for MULTILOC, F-CPMINER and TARANTULA, while OCHIAI and JACCARD need more P-EXAM score. For the remaining faults, MULTILOC dominates the other approaches. 90% of faults are located with an effort of 60% in terms of P-EXAM, where F-CPMINER and SBFL metrics need an effort greater than 85% of P-EXAM. To locate the 42 faults (100%), MULTILOC spent 90% of P-EXAM instead of 100% for the other approaches.

Our previous observations are more accentuated on fig.3b with the optimistic case. The figure shows clearly a great dominance of MULTILOC. F-CPMINER dominates the SBFL metrics until 80% of faults. Afterwards, it is with a great difficulty that F-CPMINER locates the 20% of remaining faults.

E. Statistical analysis

In order to strengthen our previous observations, we carried out a statistical test on P-EXAM and O-EXAM data. According to the fact that the data do not come from normally distributed populations and that P-EXAM and O-EXAM scores are taken from the same subjects, the *Wilcoxon Signed-Rank Test* is used [8]. We have used the one-tailed alternative hypothesis with the following null hypothesis: H_0 : Given an approach X (e.g., TARANTULA) and MULTILOC, the two groups of data are not different. For the left-tailed alternative hypothesis H_1,

EXAM	F-CPMINER	TARANTULA	OCHIAI	JACCARD
Single fault				
P-EXAM	95.91%	99.8%	70.19%	95.91%
O-EXAM	63.31%	100%	99.92%	100%
Multiple fault				
P-EXAM	99.31%	99.95%	99.96%	99.96%
O-EXAM	98.26%	99.94%	99.96%	99.92%

Table VI: Probabilities of observing H_1: MULTILOC is more efficient (One-tailed Wilcoxon Signed-Rank Test).

we state that MULTILOC is better than the given approach (i.e., less EXAM score). As we have a large number of samples for single fault (i.e., 111 programs: over 30 to be on the safe side) we have used the normal approximation with the calculated z-*value* to either accept or reject the null hypothesis.

Table VI reports the probabilities of observing H_1 on Siemens Suite using *Wilcoxon Signed-Rank Test*. For instance, the first probability given in table VI (95.91%) represents the confidence that MULTILOC is more efficient than F-CPMINER on P-EXAM score on Siemens Suite. The z-test gives us a p-value of 4.09%.

For single fault, H_1 is accepted with a high confidence. We can conclude that MULTILOC is definitely more efficient than SBFL metrics. Comparing with F-CPMINER, MULTILOC remains very competitive with a probability of 95.91% (resp. 63.31%) on P-EXAM (resp. O-EXAM). For multiple fault, we can conclude that MULTILOC is drastically efficient than SBFL metrics and F-CPMINER with a confidence over 98%.

VI. CONCLUSION

In this paper we proposed a new approach implemented in MULTILOC tool for multiple fault localization. Our approach consists of two steps: (i) extracting top-k suspicious patterns using CLOSEDPATTERN global constraint and a new measure (coined PSD for Pattern Suspiciousness Degree) while ensuring the coverage criterion; (ii) a thorough analysis of the top-k extracted patterns (i.e. ranking step) for finer-grained localization. Experiments performed on single and multiple faults programs, coming from Siemens suite benchmark, showed that our approach enables to propose a more precise localization as compared to the popular SBFL approaches and the CP-based approach F-CPMINER.

REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98, Sept 2007.

[3] P. Agarwal and A. P. Agrawal. Fault-localization techniques for software systems: a literature review. *ACM SIGSOFT Software Engineering Notes*, 39(5):5:1–5:8, 2014.

[4] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang. Evaluating the accuracy of fault localization techniques. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 76–87, 2009.

[5] M. Bekkouche, H. Collavizza, and M. Rueher. Locfaults: a new flow-driven and constraint-based error localization approach. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1773–1780, 2015.

[6] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Dellis: A data mining process for fault localization. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, USA, July 1-3, 2009*, pages 432–437, 2009.

[7] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Multiple fault localization with data mining. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*, pages 238–243, 2011.

[8] P. Dalgaard. *Introductory Statistics with R*. Statistics and Computing. Springer New York, 2008.

[9] L. De Raedt, T. Guns, and S. Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 204–212. ACM, 2008.

[10] L. De Raedt and A. Zimmermann. Constraint-based pattern set mining. In *Proceedings of the Seventh SIAM International Conference on Data Mining*, Minneapolis, Minnesota, USA, April 2007. SIAM.

[11] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, 16-19 November 2009*, pages 165–174. IEEE Computer Society, 2009.

[12] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[13] T. Guns, S. Nijssen, and L. De Raedt. Itemset mining: A constraint programming perspective. *Artificial Intelligence*, 175(12):1951–1983, 2011.

[14] T. Guns, S. Nijssen, and L. De Raedt. k-pattern set mining under constraints. *Knowledge and Data Engineering, IEEE Transactions on*, 25(2):402–418, 2013.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[16] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, New York, NY, USA, 2007. ACM.

[17] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 273–282, 2005.

[18] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 467–477, 2002.

[19] A. Kemmar, Y. Lebbah, S. Loudni, P. Boizumault, and T. Charnois. Prefix-projection global constraint and top-k approach for sequential pattern mining. *Constraints*, 22(2):265–306, 2017.

[20] N. Lazaar, Y. Lebbah, S. Loudni, M. Maamar, V. Lemière, C. Bessiere, and P. Boizumault. A global constraint for closed frequent pattern mining. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 333–349, 2016.

[21] M. Maamar, N. Lazaar, S. Loudni, and Y. Lebbah. Fault localization using itemset mining under constraints. *Autom. Softw. Eng.*, 24(2):341–368, 2017.

[22] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Inf. Syst.*, 24(1):25–46, 1999.

[23] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 465–477, 2003.

[24] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues. Learning to rank for bug report assignee recommendation. In *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*, pages 1–10, 2016.

[25] T. Uno, T. Asai, Y. Uchida, and H. Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *DS 2004*, pages 16–31, 2004.

[26] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[27] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.