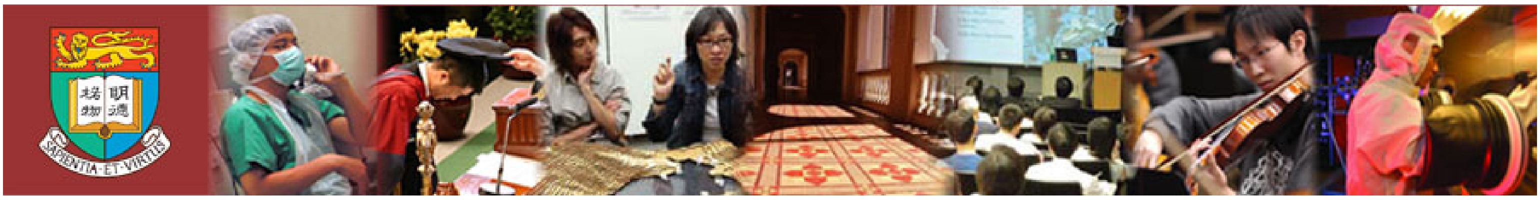


A Dynamic Fault Localization Technique with Noise Reduction for Java Programs*

Jian Xu
Department of Computer Science
Zhejiang University
Hangzhou, China
jxu@zju.edu.cn

W. K. Chan †
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

T. H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Shanping Li
Department of Computer Science
Zhejiang University
Hangzhou, China
shan@zju.edu.cn

***Abstract*—Existing fault localization techniques combine various program features and similarity coefficients with the aim of precisely assessing the similarities among the dynamic spectra of these program features to predict the locations of faults. Many such techniques estimate the probability of a particular program feature causing the observed failures. They ignore the noise introduced by the other features on the same set of executions that may lead to the observed failures. In this paper, we propose both the use of chains of key basic blocks as program features and an innovative similarity coefficient that has noise reduction effect. We have implemented our proposal in a technique known as MKBC. We have empirically evaluated MKBC using three real-life medium-sized programs with real faults. The results show that MKBC outperforms Tarantula, Jaccard, SBI, and Ochiai significantly.**

Keywords*—*fault localization; key block chain; noise reduction

I. INTRODUCTION

Fault localization is an important and yet time-consuming activity in the software development process. Coverage-based fault localization (CBFL) techniques, also known as statistical or spectrum-based techniques, have been developed to alleviate the problem. Better known examples include Jaccard [2], Tarantula [14], CBI [16], SOBER [17], and CP [26]

A typical CBFL technique involves a number of phases. It first selects a set of program features, and then collects the execution statistics of such features for both passed and failed executions. By comparing the similarities between two such sets of statistics for each feature, it estimates the extents of the program features correlated to a fault, and ranks the program features accordingly. Thus, two basic elements that affect the fault localization effectiveness in a CBFL technique are the choice of the program features and the similarity coefficient used by the technique.

Existing work has proposed to use nodes [3][15], edges [19][26], predicates [16][17], sequences of edges [8][18], sequences of conditionals in predicates [1][27], and data values [12] as program features. Moreover, the program locations to collect execution statistics for these features have been studied [3][8]. At the same time, many similarity coefficients [2][16][17][24][26] or derived coefficients [19][26][27] have been formulated.

Nonetheless, on a closer look at the experimental results of these techniques, such as [3][14][17][19][26], the mean fault localization effectiveness, even for medium-sized programs, is far below 10% when a small portion of the code (say, 2 percent) is examined. CBFL techniques are still inadequate in consistently locating program faults with a high probability.

A CBFL technique abstractly models a program as a set of features, and estimates the probability of each feature (such as fault suspiciousness) being related to the observed failures. Ideally, a program can be statically and completely partitioned into a set of equivalent classes of these features. For instance, nodes can be used as an equivalence criterion, in which case every node in a given program can be assigned to exactly one partition. Such a partitioning process may also be applied when edges or predicates are used as a criterion. Intuitively, the use of a finer-grained partitioning scheme enables a more sensitive diagnosis. However, not all kinds of models preserve such a property. For example, owing to the numerous number of possible choices, a static approach to constructing partitions such that each partition contains exactly one path or data value is impractical. Existing techniques [8][12] resolve to use dynamic partitioning strategies, which only identify a particular program feature when a program execution exhibits that feature. Methods using sequence of conditionals, such as DES [27], allow a static partitioning of predicates into sequences of conditionals. Nonetheless, to the best of our knowledge, existing approaches (such as [27]) are limited to the handling of individual compound predicates. In any case,

* This research is supported in part by grants from the Natural Science Foundation of China (project nos. 61003027 and 61073006), a grant from the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410 and 716507), and a strategy research grant of City University of Hong Kong (project no. 7008039).

† Corresponding author: Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Tel: (+852) 3442 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

1550-6002/11 $26.00 © 2011 IEEE
DOI 10.1109/QSIC.2011.32

the granularity of DES is finer than those of pure predicate-based counterparts [17]. An adaptive version of DES may consider all the predicates in a program and enumerate all possible sequences of decision values formed by these predicates, but such a simple adaptation is tedious, non-scalable, and intractable in the presence of loops.

In this paper, we propose a strategy that can statically divide a Java program into a set of partitions, each of which typically contains a series of predicates. The granularity of our method is finer than that of a node-based, edge-based, predicate-based, or the DES approach, but coarser than that of a path-based approach.

Our strategy works on an intermediate representation of the Java language, which can be described in terms of basic blocks, transition edges, and control flow graphs (CFGs) [5]. Every such block has at most one atomic predicate. We assign a sequence of such blocks into the same partition if the atomic predicate of each block is evaluated to be false. Then, a sequence of blocks to be executed will be in the same partition. We refer to such a sequence to as a *Key Block Chain* (*KBC*). Our crucial observation is that KBC provides precise information to represent the set of evaluation sequences [27] in the partition. Moreover, KBC gives clear relationships among statements, basic blocks, and predicates, which ease the tracing of a fault from the position of a predicate to a particular statement in the same block chain.

Any estimate on a particular program feature based on the result of a full path (that is, the output of an execution) is affected by the presence of other program features on the same path. Surprisingly, most existing CBFL techniques do not directly address this problem. We propose an innovative similarity coefficient that has a noise reduction effect. For each evaluation sequence, we estimate the noise factor by computing the ratio between (a) the percentage of *failed* executions that does not exercise the given evaluation sequence and (b) the percentage of *all* executions (passed or failed) that does not exercise the given evaluation sequence. We subtract this noise factor from an existing similarity coefficient. We call our technique *Minus and Key Block Chain* (*MKBC*). In this paper, we use the suspiciousness metric in Tarantula as the existing similarity coefficient. We note that the use of the suspiciousness metric merely serves as an illustration of our approach. Our approach is general.

We evaluate MKBC using a controlled experiment, and compare its effectiveness with those of Jaccard [2], SBI [24], Ochiai [2], Tarantula [14], DES [27], and CBI [16]. We use jtopas, xml-security, and ant as subject programs, which are real-life medium-sized programs and contain real faults in multiple releases. The experimental results show that, in terms of fault localization effectiveness, MKBC significantly outperforms all the other techniques studied in the experiment. For example, when checking no more than 1 percent of the code, the best peer technique (Ochiai) can find 3.45% of all faults, while MKBC finds 10.35% of them.

The main contribution of this paper is fourfold: (i) It investigates the use of KBC as feature for fault diagnosis. (ii) It proposes a formula with an explicit noise reduction term. (iii) It initiates a new fault localization technique called MKBC. (iv) It validates MKBC via an experiment.

The rest of this paper is organized as follows. Section II shows a motivating example. Section III presents our technique. Section IV presents an experimental evaluation. Section V and VI review related work and conclude this paper, respectively.

II. MOTIVATING EXAMPLE

This section uses an example to motivate the use of Key Block Chains for fault localization. Fig. 1 shows the program code excerpted from a faulty version of the program ant, downloaded from the Software-artifact Infrastructure Repository (SIR) [9]. The functionality of this code excerpt is to translate the path of a file from OS-format into VM-format. A fault exists on statement S_2, where the second parameter of the method path.indexOf( ) should be 1 (rather than 0). In this example, exercising S_2 followed by S_4 triggers a failure.

1) Jimple

Jimple is an intermediate representation [20][21] of Java. We observe that Jimple has several desirable properties to support fault localization. First, Jimple always normalizes every compound Boolean expression into atomic Boolean expressions, each of which resides in exactly one basic block. Second, each basic block contains at most one atomic Boolean expression. Third, mapping a Boolean expression in Jimple code to its corresponding statement in Java code is easy.

The Jimple code of the program excerpt[1] and the control flow graph (CFG) based on Jimple code are also shown in Fig. 1. We observe that the compound predicate at S_6 is split into two basic blocks. Also, a basic block B_2 contains the statements S_2 and S_3. Its preceding block is B_1 and its succeeding blocks are B_3 and B_4. The connections between a basic block and its preceding/succeeding basic blocks are explicitly captured in Jimple representation. For ease of presentation, we add a dummy block B_8 (marked as a dashed box) and four edges (marked as dashed arrows) to make the mapping between the Jimple code and the CFG more explicit.

We further denote the predicate in a block B_i by P_i and the corresponding predicate at a statement S_i by p_i. For instance, we use P_1 to denote predicate for B_1 and p_3 to denote predicate for S_3.

2) Test Cases and Executions

Fig. 1 shows 10 sample test cases, together with their pass/fail statuses. The statement- and block-execution information is also shown in the figure. A cell filled with "■" indicates that the corresponding statement or block is exercised by the execution of that test case. A cell filled with "▲" indicates that the corresponding statement or block is only partially exercised. Let us take the fourth test case T_4 as an example. When the program executes T_4,

[1] Note that, to realize the streamline design [20][21] in Jimple, the source code has been transformed with some branches switched without altering the program behavior. For example, the condition "`index == -1`" *in* S_3 is changed to "`index != -1`" with the corresponding branches swapped.

Java Statements and Jimple Statements	Test Cases	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Jaccard [2]		SBI [24]		Tarantula [14]		CBI [16]		HOLMES [8]		Minus (this paper)		MKBC– (this paper)		MKBC (this paper)	
	Pass/Fail	F	P	F	P	P	P	P	P	P	P	sus	r	sus	r	sus	r	sus	r	sus	r	sus	r	sus	r	conf	r
`if (isAbsolute) {`	S_1	■	■	■	■	■	■	■	■	■	■	0.2	6	0.2	7	0.5	6	N/A	5	N/A	8	0.5	3	0.44	4	–0.09	4
`index = path.indexOf(File.separatorChar, 0);`	S_2	■	■	■		■	■		■	■	■	0.25	**5**	0.25	**5**	0.57	**5**	N/A	**7**	N/A	**8**	**0.57**	**2**	**0.44**	**4**	**0.16**	**3**
`if (index == -1) {`	S_3	■	■	■		■	■		■	■	■	0.25	5	0.25	5	0.57	5	N/A	6	N/A	8	0.57	2	0.44	4	0.16	3
`return path.substring(1) + ":[000000]";` `} else {`	S_4	■								■		0.33	1	0.5	1	0.8	1	N/A	7	N/A	8	0.44	4	0.44	4	0.44	1
`device = path.substring(1, index++); }}`	S_5		■	■		■	■		■		■	0.17	7	0.22	6	0.44	7	N/A	5	N/A		–0.13	7	–0.13	8	N/A	8
`...`																											
`if (!isAbsolute && directory != null) {`	S_6		■	■	▲	▲	■	▲	■		■	0.11	8	0.13	8	0.36	8	N/A	1	N/A	8	–0.44	8	0.27	7	–0.15	7
`directory.trim();`	S_7		■	■					■			0.25	5	0.33	3	0.66	3	N/A	5	N/A	8	0.27	6	0.27	7	0.27	6
`directory.insert(0, '.'); }`	S_8		■	■					■			0.25	5	0.33	3	0.66	3	N/A	5	N/A	8	0.27	6	0.27	7	0.27	6

Java Statements and Jimple Statements	Test Cases	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}

`Block 1:[preds:] [succs: 2 5]`
`...`
`1: if isAbsolute == 0 goto if isAbsolute!=0 ...`
	B_1	■	■	■	■	■	■	■	■	■	■

`Block 2:[preds: 1] [succs: 3 4]`
`2: $c0 = <java.io.File: char separatorChar>`
`2: index = virtualinvoke path. ...`
`3: if index != -1 goto index = index + 1`
	B_2	■	■	■		■	■		■	■	■

`Block 3:[preds: 2] [succs:]`
`...`
`4: return $r3`
	B_3	■								■	

`Block 4:[preds: 2] [succs: 5]`
`5: index = index + 1`
`5: virtualinvoke path. ...`
`...`
	B_4		■	■		■	■		■		■

`Block 5:[preds: 1 4] [succs: 6 8]`
`6: if isAbsolute != 0 goto return`
	B_5		■	■	■	■	■	■	■		■

`Block 6:[preds: 5] [succs: 7 8]`
`6: if directory == null goto return`
	B_6		■	■			■		■		■

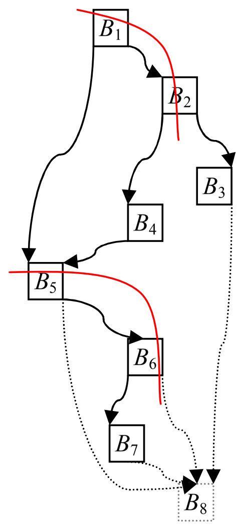
`Block 7:[preds: 6] [succs: 8]`
`7: virtualinvoke directory. ...`
	B_7		■	■					■		

`Block 8:[preds: 5 6 7] [succs: ]`
`return`											

Paths of HOLMES		
1. $B_1 \rightarrow B_5$	N/A	7
2. $B_1 \rightarrow B_5 \rightarrow B_6$	N/A	7
3. $B_1 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7$	N/A	7
4. $B_1 \rightarrow B_2 \rightarrow B_3$	0.21	2
5. $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$	N/A	7
6. $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5 \rightarrow B_6$	N/A	7
7. $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7$	0.21	2

Predicates and Paths of MKBC			
N_1 $\{P_1, P_2\}$			
1. $B_1 \rightarrow B_5$	$P(1, 5)$	–0.57	6
2. $B_1 \rightarrow B_2 \rightarrow B_3$	$P(1, 2, 3)$	0.44	
3. $B_1 \rightarrow B_2 \rightarrow B_4$	$P(1, 2, 4)$	–0.13	3
N_2 $\{P_5, P_6\}$			
4. $B_5 \rightarrow B_8$	$P(5, 8)$	–0.57	6
5. $B_5 \rightarrow B_6 \rightarrow B_8$	$P(5, 6, 8)$	–0.57	6
6. $B_5 \rightarrow B_6 \rightarrow B_7$	$P(5, 6, 7)$	0.27	2

Predicates for CBI		
1. p_1	0.05	2
2. p_3	0	3
3. p_6	0.21	1



	Jaccard [2]	SBI [24]	Tarantula [14]	CBI [16]	HOLMES [8]	Minus (this paper)	MKBC– (this paper)	MKBC (this paper)
% of code examined to locate fault	**62.5%**	**62.5%**	**62.5%**	**87.5%**	**100%**	**25%**	**50%**	**37.5%**

Figure 1. A faulty version of program ant and effectiveness comparison of different fault localization techniques (sus.: suspiciousness of a statement/block/path being related to a fault; r.: ranking of a statement/block/path; conf.: confidence with respect to tie breaking).

statement S_1 is exercised, S_6 is partially exercised, and basic blocks B_1 and B_5 are exercised. The compound predicate of S_6 is split into two basic blocks B_5 and B_6 at the Jimple level, where B_5 represents the first conditional of S_6, which is exercised by T_4; B_6 represents the second conditional of S_6, which is not exercised. Consequently, we mark S_6 by "▲" in the T_4 column. Other test cases can be interpreted similarly.

3) Peer Techniques

We apply techniques Jaccard [2], SBI [24], Tarantula [14], CBI [16], and HOLMES [8] on the same example and compute, for each statement or block, the corresponding suspiciousness scores and their ranks. They are shown in the "sus" and "r" columns, respectively. By calculating the value of expense [24] for each technique, their effectiveness in locating the fault in S_2 is measured by the percentage of code that must be examined (as recommended by the expense technique) to include S_2. The value of expense is shown in the "% of code examined to local fault" row.

None of Tarantula, Jaccard, and SBI can rank S_2 as the most suspicious statement. Nonetheless, statements S_6, S_7, and S_8 are mistakenly deemed as highly suspicious by these techniques. Intuitively, these statements are considered dubious because they are closest to the fault and, at the same time, have been executed by both failed and passed test cases.

CBI also fails to assign high suspiciousness values to the predicates p_1 and p_2, which are the predicates closest to the faulty position. Apparently, this is because these predicates have been evaluated to be true by quite a number of passed test cases. CBI even ranks other predicates such as p_6 higher than the predicates p_1 and p_2. HOLMES uses paths as a feature to locate faults. In the example, the 10 test cases result in seven paths as shown in the "Paths of HOLMES" section of Fig. 1. We observe that the same predicate may appear in multiple paths, and a path may contain many predicates. Obviously, many non-faulty statements need to be examined before the most fault-relevant paths can be identified by HOLMES.

4) A Sketch of Our Approach

Our approach consists of three steps. We first construct KBC predicates as a program feature, then use a similarity coefficient formula to estimate the fault suspiciousness, and finally map the feature suspiciousness into block suspiciousness.

We traverse the Jimple code block by block, starting from the first block. We iteratively mark every block containing a predicate until we encounter a block whose last statement is not a predicate. We link all the marked blocks during this search to form a chain, and clear all the marks. We then repeat the search. In this way, we partition the Jimple code into a number of sections, and refer to each section as a Key Block Chain (KBC). We further extract the set of predicates from each KBC as a program feature for the KBC, and refer to each feature as a KBC predicate. In Fig. 1, for example, we start the search from B_1. Because the last statement in B_1 is a predicate, we mark B_1 and continue to traverse B_2. B_2 is also marked because it also contains a predicate. We then visit B_3, which contains no predicate. As such, we link blocks B_1 and B_2 to form a KBC. We then clear the marks in B_1 and B_2, and repeat the search at the next block B_4. Note that we do not include B_3 into the formed KBC because it contains no predicate. Finally, we construct another KBC by linking B_5 and B_6. We therefore obtain two sets of KBC predicates, $\{P_1, P_2\}$ and $\{P_5, P_6\}$.

A KBC predicate may contain several atomic Boolean expressions. We use them to construct evaluation sequences [27] according to their decision results. Given a KBC N_i, let $P(N_{i,j})$ denote the sub-path j of N_i. We compute its suspiciousness score using a formula in the form of "α – β". The first term α calculates the ratio between (i) the percentage of *failed* executions that exercise $P(N_{i,j})$ and (ii) the percentage of *all* executions (passed or failed) that exercise $P(N_{i,j})$. Thus, α estimates the probability of exercising $P(N_{i,j})$ when failures occur. We use the suspiciousness metric $\frac{\%\text{failed}(P(N_{i,j}))}{\%\text{failed}(P(N_{i,j}))+\%\text{passed}(P(N_{i,j}))}$ in Tarantula as the similarity coefficient to compute α. The second term β calculates the ratio between (iii) the percentage of *failed* executions that do not exercise $P(N_{i,j})$ and (iv) the percentage of *all* executions that do not exercised $P(N_{i,j})$. In this way, β estimates the probability of not exercising $P(N_{i,j})$ when failures occur. We use the term $\frac{1-\%\text{failed}(P(N_{i,j}))}{1-\%\text{failed}(P(N_{i,j}))+1-\%\text{passed}(P(N_{i,j}))}$ to compute β. As a result, the formula α – β is a more precise estimation of how much a sub-path $P(N_{i,j})$ relates to the observed failures.

Surprisingly, many existing fault suspiciousness formulas ignore the second term. Attributing it to the nature of noise reduction, we call our formula *Minus*. We note that by substituting $P(N_{i,j})$ for a statement S_i into the formula, it can be adapted to work at the statement level. For example, the suspiciousness of statement S_4 can be computed as $\frac{1/2}{1/2+1/8} - \frac{1-1/2}{1-1/2+1-1/8} = 0.80 - 0.36 = 0.44$. The term 0.80 is due to Tarantula while the term 0.36 is the noise reduction using Minus, so that the suspiciousness score of the correct statement S_4 is reduced accordingly.

We recall that a block may reside on more than one sub-path under the same KBC predicate. We define the suspiciousness of a block to be the maximal suspiciousness of all the sub-paths of the KBC predicate that the block resides on. Moreover, we use the mean value of these suspiciousness scores as the tie breaker value for the block in case it is in tie with any other block.

We call our technique Minus and Key Block Chain (MKBC). For example, the suspiciousness of statement S_2 is finally calculated as (sus = 0.44, conf = 0.16) and (sus = 0.44) by MKBC with and without tie breaking, respectively. The details are given in Section III.

In the example, we compare five peer techniques with Minus (which adopts our formula to work on statements), MKBC– (which adopts our method but works at the KBC predicate level), and MKBC (with our tie breaking strategy). As shown in Fig. 1, the expense values [24] of the various techniques are (from left) 62.5%, 62.5%, 62.5%, 87.5%, 100%, 25%, 50%, and 37.5%, respectively. The example shows that the use of our formula at either the statement level or the KBC predicate level, either with or without tie breaking, can be promising.

III. Our Model

A. Preliminaries

Given a program, we use $G(P) = \langle B, E\rangle$ to denote the control flow graph (CFG) of its Jimple code, where $B = \langle B_1, B_2, \ldots, B_n\rangle$ is the set of basic blocks [5]. Let $T = \langle t_1, t_2, \ldots, t_u\rangle$ be a set of passed test cases, and $T' = \langle t_1', t_2', \ldots, t_v'\rangle$ be a set of failed test cases.

B. Key Block Chain Model

Our model consists of three major steps: the construction of KBC predicates as a program feature, the calculation of suspiciousness scores for the KBC predicates, and the mapping of the suspiciousness scores to the blocks.

1) Constructing KBC Predicates as Program Feature

As illustrated in the motivating example, we start from the first block B_1 in the Jimple code, and create an empty sequence *s*. We then conduct the following general procedure: If a block B_i contains a predicate, we append B_i to *s*. If B_i contains no predicate, then we output *s* as a KBC and reset *s* to empty. After the checking, we increase the counter *i* by 1, and repeat the procedure until the last block has been processed. Since the process is straightforward, we do not include the formal algorithm in this section.

Every KBC contains a sequence of blocks, each of which contains exactly one atomic predicate. This sequence of atomic predicates is called a KBC predicate. According to Jimple semantics, if such an atomic predicate in a block is evaluated to be true, the block following the atomic predicate (which is a block in the same KBC) will not be executed. Moreover, the execution will jump to a succeeding block (defined by the "[succ]" annotation) of that block. For each KBC, by enumerating the possible underlying decision value of each atomic predicate, the corresponding KBC predicate can be mapped to a set of sub-paths in the program. We use the notation $P(N_{i,j})$ to denote such a sub-path *j* with respect to the KBC N_i. In Fig. 1, for example, the KBC N_1, which contains the KBC predicates (P_1, P_2), may be resolved into three sub-paths $P(1, 5)$, $P(1, 2, 3)$, and $P(1, 2, 4)$, where $P(1, 5)$ denotes the sub-path $B_1 \rightarrow B_5$ and so on.

2) Calculating Suspiciousness Scores for Features

To calculate the suspiciousness score of $P(N_{i,j})$, which is denoted by $\theta(P(N_{i,j}))$, we propose the following equation:

$$\theta\left(P(N_{i,j})\right) = \frac{\%\,\text{failed}\left(P(N_i)\right)}{\%\,\text{failed}\left(P(N_{i,j})\right)+\%\,\text{passed}\left(P(N_{i,j})\right)} - \frac{1-\%\,\text{failed}\left(P(N_{i,j})\right)}{1-\%\,\text{failed}\left(P(N_{i,j})\right)+1-\%\,\text{passed}\left(P(N_i,j)\right)}. \qquad (1)$$

Equation (1) is in the form of "α – β". The first term α calculates the ratio between (i) the percentage *failed* executions that exercise $P(N_{i,j})$ and (ii) the percentage *all* executions (passed or failed) that exercise $P(N_{i,j})$. This roughly estimates the extent that the exercising of $P(N_i)$ causes the observed failures. Similarly, the second term β calculates the ratio between (iii) the percentage of *failed* executions that do not exercise $P(N_{i,j})$ and (iv) the percentage *all* executions that do not exercise $P(N_{i,j})$. In this way, the expression α – β is a more precise estimation of the extent that the exercising of $P(N_{i,j})$ relates to the observed failures. This expression represents a change in failure probability from not executing a path to executing a path, aiming at reducing the effect of the tailing parts of the first term (that is, exercising some other sub-paths when we measure the probability based on $P(N_{i,j})$). Owing to the noise reduction involved, we refer to equation (1) as Minus.

Consider $P(1, 2, 4)$ in Fig. 1 as an example. %failed($P(1, 2, 4)$ = 0.5 and %passed($P(1, 2, 4)$ = 0.625. Hence, α = 0.5 / (0.5 + 0.625) = 0.44 and β = (1 – 0.5) / (1 – 0.5 + 1 – 0.625) = 0.57, giving $\theta(P(1, 2, 4))$ = α – β = –0.13.

3) Mapping to Suspiciousness Scores to Blocks

A block may appear in multiple sub-paths. We therefore define the suspiciousness of a block in a sub-path to be the maximum of the suspiciousness scores of all the sub-paths of the same KBC that the block resides on, thus:

$$sus(B_i) = \max\left\{\theta\left(P(N_{i,j})\right)\right\} \qquad (2)$$

We choose to use the maximum operator because we aim to keep a close relationship between the block and the most effective sub-path that the block resides on.

Finally, we give a tie breaking strategy to resolve tie cases [14]. We use the mean suspiciousness value of all the sub-paths of the same KBC that the block resides on as the tie breaker value of the block. The formula is

$$conf(B_i) = \overline{\sum_{B_\iota \in N_{J,k}} \theta\left(P(N_{J,k})\right)} \qquad (3)$$

For example, the tie breaker value of B_2 is calculated as conf(B_2) = (0.44 + (–0.13)) / 2 = 0.16.

4) Further Issues

After obtaining the suspiciousness score for every block, it is simple to compute the suspiciousness score for every statement. A statement in the source code may be split into multiple Jimple statements, which may belong to different blocks. We choose the highest suspicious value of all those Jimple statements to be the suspiciousness value of the statement at the source code. Finally, we may assign ranks to statements. Like previous work, the rank of a statement is defined as the total number of statements whose suspiciousness values are higher or equal to it.

Sometimes, a function in a program may contain no predicate. In that case, we simply add a dummy predicate that is always evaluated to be false at the end of the first block, so that the faults in the function will not be overlooked. We also note that we need only check the last statement of each block to determine whether it is a predicate. This search can be performed in $O(n)$ time, where n is the number of blocks in the Jimple code.

IV. Controlled Experiment

In this section, we report a controlled experiment that evaluates the effectiveness of our techniques and compare them with peer techniques.

A. Experimental Setup

1) Subject Programs

The experiment uses three real-life programs, namely, jtopas, xml-security, and ant, as subject programs. We have downloaded them (including all the faulty versions and associated test suites) from the SIR site [9]. Table I shows the descriptive statistics of each subject program, including the versions, the program size (in LOC), the number of faulty versions, and the size of the associated test pool. Following [14], we execute each version with each test case, and input the entire set of executions to each technique, which will be described below.

TABLE I. DESCRIPTIVE STATISTICS OF SUBJECT PROGRAMS

	Real-Life Versions	Program Description	LOC	No. of Versions	No. of Test Cases
jtopas	0.4 – 0.6	Text parser	5400	25	207
xml-security	1.0.4 – 1.0.71	XML signature and encryption	16800	49	94
ant	1.6 beta	Tool building	80500	12	830
Total				86	1131

Following the documentation of SIR [9] and previous experiments [1][10][17][26], we exclude the versions whose faults cannot be revealed by any test case. This is because both our techniques and peer techniques do comparisons on profiling produced by failed test cases and passed test cases. In addition, several old program versions such as ant versions prior to 1.6, which are based on JDK 1.4, are excluded. Our instrumentation tool, implemented on Soot [20] version 2.3.0 running on JDK 1.6, does not support them. We finally use all the remaining 86 faulty versions, as shown in Table I, in the experiment.

2) Experimental Environment

Our experiments were run on a Ubuntu 8.04 Desktop system serving a VMware virtual machine with a configuration of a single Intel® Core™ Duo 2.66 GHz CPU, 512 MB memory, and 20 GB hard disk. Our tool is developed on top of Soot version 2.3.0 [20]. All the programs and tools are compiled with JDK 1.6. Test cases are managed by the JUnit framework version 3. All the work was run automatically using bash scripts.

3) Peer Techniques

Because our techniques can be adopted to operate at the statement level, we select four statement-level peer techniques to compare with in the experiment. Tarantula [14] with its tie breaking strategy is used because the use of a tie breaking strategy has been shown to give Tarantula a better result owing to a finer division of ranks. Jaccard [2] and Ochiai [2], with strong mathematical theory, are shown to be effective in fault localization. SBI [24] has been adapted from CBI [16] to statement level. In short, all the four selected techniques have been validated to be excellent fault localization techniques by existing research.

4) Effectiveness Metric

Each of these techniques produces a ranked list of all the executed statements in descending order of their computed suspiciousness values. The *rank* of a statement is defined as the largest number of statements from the top of the list until and including this statement and all the next following statements sharing the same suspicious score with it.

Previous work [24] defines the *expense* metric as the ratio between the rank of the faulty statement and the total number of executable statements. We consider, however, that the use of the number of *executed* statements as the denominator in the expense formula is more suitable because other unrelated statements do not need to be checked in practice.

At the same time, if a fault is on a non-executable statement (such as a code omission fault), the use of dynamic execution information cannot help locate the fault directly. Following [12], we mark the directly affected statement or an adjacent executable statement as the faulty position, followed by applying the expense metric.

5) Experimental Procedure

We compare Jaccard, Ochiai, SBI, and Tarantula with our techniques. We use the data created from the techniques described in papers [2], [2], [24], and [14], respectively, and compare them with MKBC and its variant Minus. Minus is an adapted technique from MKBC, where we use statements at the source code level (instead of KBCs) as the program feature. The aim of the experiment is to study whether MKBC is an effective technique to locate faults.

In the experiment, we input the entire test pool for each version to each technique studied and measure their expense values accordingly.

B. Data Analysis

1) Overall Results

Fig. 2 shows a comparison of overall effectiveness. The x-axis indicates the percentage of code that needs to be examined to locate the fault. We also refer to it as the code examination effort in this paper. The y-axis indicates the percentage of faults located. Take the curve of MKBC in Fig. 2 as an illustration. MKBC locates 35.63% of all the faults by examining no more than 10 percent of the code in a faulty version. The curves of Jaccard, Ochiai, SBI, Tarantula, and Minus can be interpreted similarly. Note that the curves of Jaccard and Ochiai are almost completely overlapping.

We observe that MKBC can locate many more faults than all the others in the code examination range between 0 to 90 percent, and Minus is almost as effective as (and hence has little advantage over) the other techniques in most ranges. The latter shows that merely adopting the formula of equation (1) has marginal advantages or disadvantages over existing techniques.

Table II shows the detailed results in Fig. 2. It indicates that, when no more than 1 percent of the code can afford to be examined, Jaccard, Ochiai, Tarantula, and Minus locates 3.45% of the faults, while SBI does not locate any fault. MKBC is the best among all the techniques in the experiment, locating 10.35% of the faults.

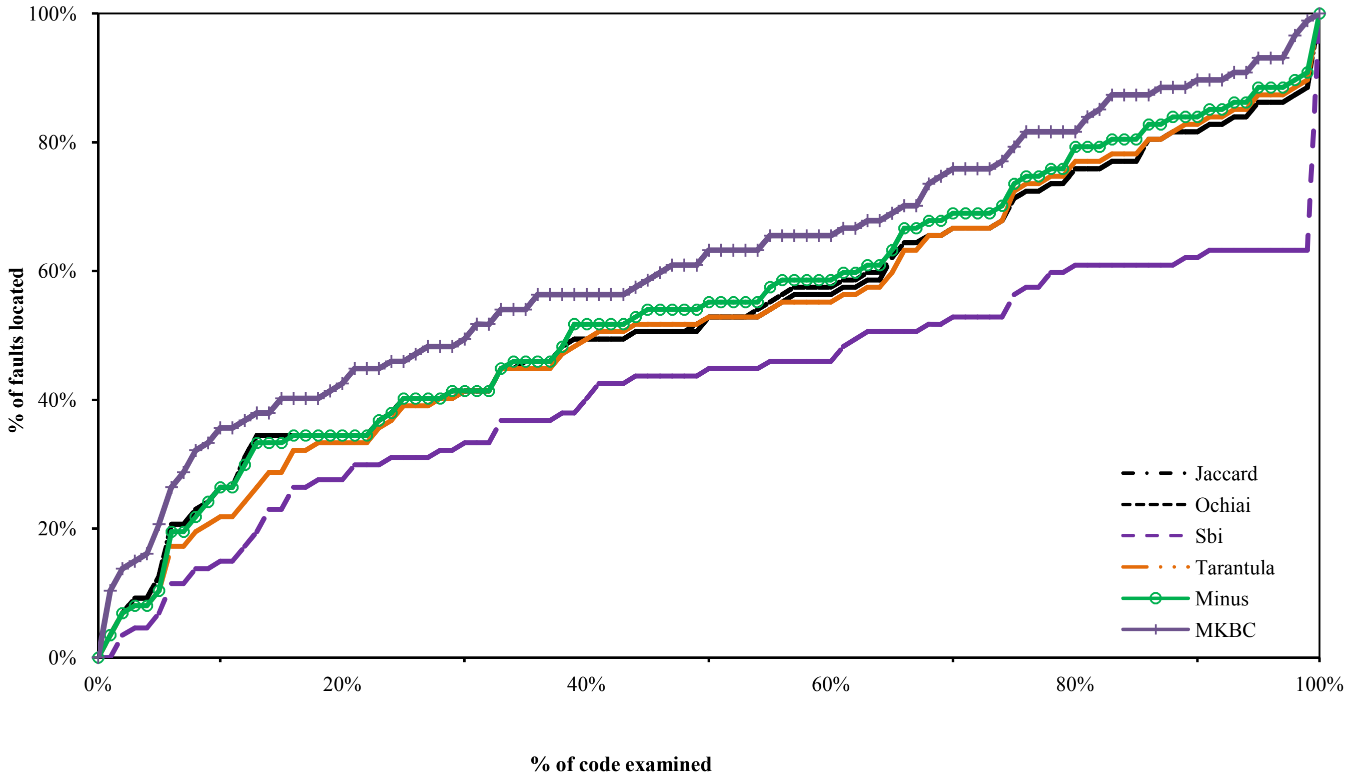


Figure 2. Overall effectiveness comparison.

Table III presents the minimums (min), maximums (max), medians, means, and standard derivations (stdev) of the effectiveness of these techniques on the 86 faulty versions. Take the first row as an illustration. MKBC can locate a fault by examining 0.32% of the code in the best case, while others need to examine more. Overall, the table shows that MKBC is the best among the six techniques. The results show that the use of KBC as a feature is very promising.

TABLE II. OVERALL EFFECTIVENESS

expense	***MKBC***	*Minus*	Jaccard [2]	Ochiai [2]	SBI [24]	Tarantula [14]
1%	***10.35%***	*3.45%*	3.45%	3.45%	0%	3.45%
2%	***13.79%***	*6.90%*	6.90%	6.90%	3.45%	6.90%
5%	***20.70%***	*10.34%*	12.64%	12.64%	6.90%	10.34%
7%	***28.76%***	*19.54%*	20.70%	20.70%	11.49%	17.24%
10%	***35.63%***	*26.44%*	26.44%	26.44%	14.94%	21.84%
20%	***42.53%***	*34.48%*	34.48%	34.48%	27.59%	33.33%
30%	***49.43%***	*41.38%*	41.38%	41.38%	33.33%	41.38%
40%	***56.32%***	*51.72%*	49.43%	49.43%	40.23%	49.43%
50%	***63.22%***	*55.17%*	52.87%	52.87%	44.83%	52.87%
60%	***65.52%***	*58.62%*	56.32%	57.47%	45.98%	55.17%
70%	***75.86%***	*68.97%*	66.67%	66.67%	52.87%	66.67%
80%	***81.61%***	*79.31%*	75.86%	75.86%	60.92%	77.01%
90%	***89.66%***	*83.91%*	81.61%	81.61%	62.07%	82.76%
100%	***100%***	*100%*	100%	100%	100%	100%

To further study the relative merits of our techniques, we compare the effectiveness of MKBC and Minus with every peer technique. The results are shown in Table IV.

Take the cell in column "MKBC – Tarantula" and row "< –5%" as an example. It states that, for 34 (39.53%) of the 86 faulty versions, the code examination effort when using MKBC to locate a fault is less than that when using Tarantula by more than 5%. Similarly, for the row "> 5%", for only 3 (3.49%) of the 86 versions, the code examination effort of MKBC is greater than that of Tarantula by more than 5%. For 49 (56.98%) of the faulty versions, the effectiveness between MKBC and Tarantula cannot be distinguished at a 5% significance level.

We therefore conclude that, on average, at a 5% significance level, the probability of MKBC performing better than Tarantula on the subject programs is higher than that of Tarantula performing better than MKBC. We further vary the significance level from 5% to 1% and 10% to check the sensitivity of our result. It shows that the probability of MKBC performing better than peer techniques is consistently higher than that in the other way round.

2) Results on Individual Subject Programs

We further compare the effectiveness of Minus and MKBC with peer techniques on each individual subject program. Fig. 3 shows the results. Each plot can be interpreted similar to Fig. 2.

For example, if 10 percent of the code is examined, MKBC can locate faults in 56.00%, 28.00%, and 25%, and

TABLE III. Statistics of Overall Effectiveness

	MKBC	*Minus*	Jaccard [2]	Ochiai [2]	SBI [24]	Tarantula [14]
min	***0.32%***	*0.65%*	0.65%	0.65%	1.02%	0.65%
max	***100.00%***	*100.00%*	100.00%	100.00%	100.00%	100.00%
median	***26.83%***	*38.12%*	38.54%	38.54%	61.87%	39.78%
mean	***38.72%***	*45.39%*	46.87%	46.73%	58.30%	47.39%
stdev	***34.28%***	*35.17%*	36.28%	36.24%	38.37%	35.37%

TABLE IV. Comparisons in Overall Effectiveness

	Difference (Percentage Difference) in Overall Effectiveness				
	MKBC – Minus	***MKBC – Jaccard* [2]**	***MKBC – Ochiai* [2]**	***MKBC – SBI* [24]**	***MKBC – Tarantula* [14]**
< –1%	43 (50.00%)	47 (54.65%)	47 (54.65%)	59 (68.60%)	48 (55.81%)
–1% to 1%	36 (41.86%)	33 (38.37%)	33 (38.37%)	25 (29.07%)	34 (39.53%)
> 1%	7 (8.14%)	6 (6.98%)	6 (6.98%)	2 (2.33%)	4 (4.65%)
< –5%	26 (30.23%)	28 (32.56%)	28 (32.56%)	49 (56.98%)	34 (39.53%)
–5% to 5%	56 (65.12%)	53 (61.63%)	53 (61.63%)	35 (40.70%)	49 (56.98%)
> 5%	4 (4.65%)	5 (5.81%)	5 (5.81%)	2 (2.33%)	3 (3.49%)
< –10%	20 (23.26%)	22 (25.58%)	22 (25.58%)	36 (41.86%)	21 (24.42%)
–10% to 10%	64 (74.42%)	61 (70.93%)	61 (70.93%)	48 (55.81%)	63 (73.26%)
> 10%	2 (2.33%)	3 (3.49%)	3 (3.49%)	2 (2.33%)	2 (2.33%)

of the faulty versions of the program xml-security, jtopas, and ant, respectively; while Minus can locate 48.00%, 18.00%, and 16.7% of faults, respectively. Tarantula can only locate 36% and 16%, and 16.7% respectively. The results of Jaccard, Ochiai, and SBI can be interpreted similarly.

We find that MKBC performs consistently better than peer techniques on xml-security and ant. For jtopas, there is only a small code examination range that MKBC is less effective than Minus or Ochiai. Note that in the initial code examination range, MKBC still performs better than all the other techniques studied. We conclude that MKBC is promising on every subject.

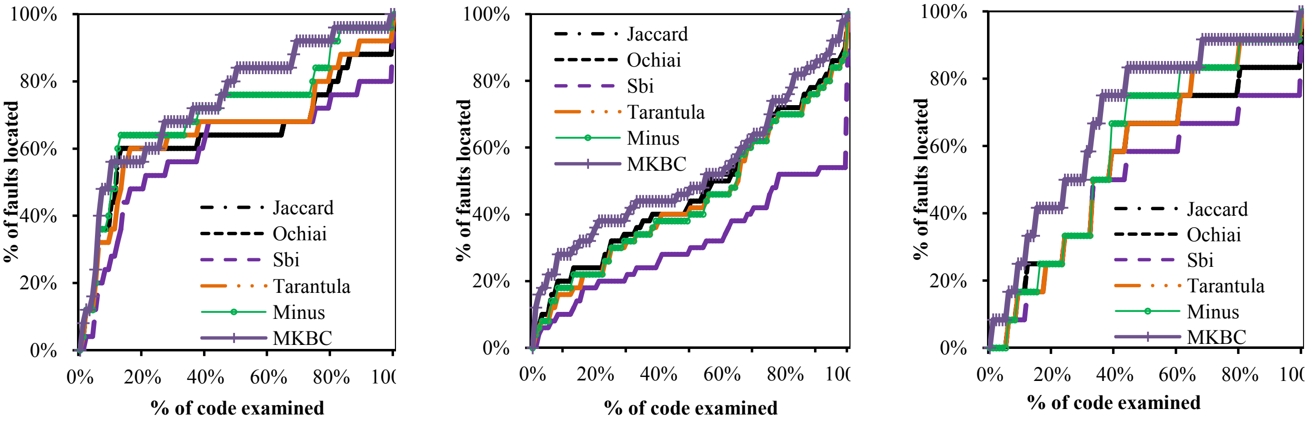
TABLE V. Comparisons of Run Time Costs

	Minus /Jaccard	**Minus /Ochiai**	**Minus /SBI**	**Minus /Tarantula**	**Minus /MKBC**
> 1.1	11	14	12	15	0
0.9 to 1.1	46	53	57	61	0
< 0.9	29	19	17	10	88
mean	0.94	0.98	0.99	1.01	0.37
	MKBC /Jaccard	**MKBC /Ochiai**	**MKBC /SBI**	**MKBC /Tarantula**	**MKBC /Minus**
> 1.1	78	82	85	86	86
0.9 to 1.1	7	4	1	0	0
< 0.9	1	0	0	1	1
mean	2.94	3.05	3.06	3.08	3.09

C. Performance Analyses

We compare MKBC and Minus with peer techniques in terms of the run time from profiling to the output of a ranked list of statements. We run every version 20 times and then use the average time as the result, which is shown in Table V.

Take the comparison between Minus and Tarantula as an example. Let T_{Minus} denotes the run time of Minus and $T_{\text{Tarantula}}$ denote that of Tarantula. If the ratio T_{Minus} / $T_{\text{Tarantula}}$ is more than 1, the performance of Minus is worse than Tarantula. Consider the cells in column "Minus / Tarantula" and rows "> 1.1", "0.9 to 1.1", "< 0.9", and "mean". They indicate that (a) the run times from profiling to ranking by Minus are greater than those by Tarantula in 15 faulty versions by more than 10%, (b) they cannot be distinguished in 61 faulty versions at a 10% significance level of 10%, (c) the run times by Minus are less than those by Tarantula in 10 faulty versions by less than 10%, and (d) the mean ratio of run times is 1.01. The comparisons among other techniques can be interpreted similarly.



(a) jtopas (b) xml-security (c) ant

Figure 3. Effectiveness on individual programs.

Considering Tables III to V together, we conclude that Minus and MKBC can be used in different situations. If we want to locate faults quickly, Minus is a better choice. Its run time from profiling to ranking is about the same as other techniques but can locate a fault by examining less number of statements than existing peer techniques. On the other hand, MKBC is much slower than peer techniques, with a run time almost three times those of the others. However, it can locate 6% to 8% more faults when examining the same percentage of code.

D. Threats to Validity and Discussions

We use Soot to insert probes into the Java bytecode. Soot gives a good solution for specific Java features such as exception handling. Previous work [11][25] has investigated on this topic, as exception information in run time contains plenty of error information, thus providing good support to fault localization. In this paper, we consider exception handling in programs as normal control flow because Soot is able to transform a Java program into Jimple and still maintain the exception handling structures. Hence, if faults are located in these "Catch" blocks, the approach in this paper can still find them.

An interesting strategy is that, since we find ant 1.6 beta to be our test subject, we also use the latest ant as a build tool.

Soot 2.3.0 is based on Java 1.5 or higher, but some of our subject programs are originally based on Java 1.4. We need to modify these subjects so that they are compatible to Java 1.5 or higher. It entails some modifications, and we have carefully reviewed the conversion.

The strategy we use to construct a KBC is only one possible solution among many. Other strategies are also feasible. We briefly discuss some possible extensions of our work. The first strategy is to identify blocks containing predicates that are as long as possible. This strategy is close to the full path tracking idea used in HOLMES [8]. Such a strategy, however, requires a search of the longest path from a graph, which takes more than $O(n)$ time. A second strategy is to identify blocks containing predicates and use a random sequence of blocks to construct a chain. Yet another strategy is to identify sequences of blocks within certain lengths and split a long chain into several shorter ones. An optimal length of a block chain is hard to be determined. Moreover, one obvious limitation of the last two strategies is that they may knot irrelevant blocks together.

Another important prospect is that KBC can be applied to any program entity level. In computing, compilers usually decompose programs into their basic blocks as the first step in the analysis process. Other languages can also have streamline representations like Jimple for Java. Applying KBC to them helps locate faults written in these programs.

V. Related Work

Tarantula [14] uses the proportions of failed or passed executions to calculate the suspiciousness of every statement. Jones et al. [15] further use Tarantula to explore how to assist multiple developers to debug a program in parallel. CBI [16] uses predicates as fault indicators to locate faults. They rank the predicates P according to the probability that the program under study will fail when P is observed to be true. Arumuga Nainar et al. [1] use compound Boolean predicates based on CBI to locate faults. Zhang et al. [27] show experimentally that short-circuit rules in the evaluation of Boolean expressions may significantly affect the effectiveness of predicate-based techniques, and propose DES [27] accordingly. HOLMES [8] uses a full path as a fault predicator and proposes an iterative way to reduce the cost of profiling. Jiang and Su [13] propose another way to generate faulty control flow paths from bug predicators by using a depth-first search to greedily find paths that connect as many fault indictors as possible and reducing unlikely faulty paths to generate fault-related paths interactively. Zhang et al. [26] develop a CP approach that captures the propagation of infected program states through edges in a control flow graph. CP associates suspiciousness scores of control flow edges to suspiciousness scores of basic blocks to locate faults. Santelices et al. [19] investigate different program entities (such as statements, edges, and du-pairs). They show that integrated results of different entities may perform better than individual ones. Yilmaz et al. [23] leverage time spectra as abstractions of program executions. They use them for functional correctness debugging by identifying program segments that take a "suspicious" amount of time to execute.

Selecting a set of good test cases is also an important way to improve the effectiveness of fault localization. Baudry et al. [6] identify a property known as dynamic basic block to improve the accuracy of a diagnosis algorithm. Cellier [7] combines association rules and formal concept analysis to figuring out whether a failure is due to one statement or multiple ones.

Abreu et al. [1] propose a new approach to locating faults in multi-fault programs. We believe that our future work can incorporate this aspect because the Minus formula appears to have the potential to insulate the interferences among multiple faults.

VI. Conclusion

Various techniques have been proposed in existing coverage-based fault-localization research. They choose different program features, combine some of them, or enrich individual features with context-sensitive information. They then compute a ranked list of the suspiciousness of the program features using different models. It is important to choose suitable program features and models, as well as consider the trade-off among them and their computing costs. On the other hand, in existing problem settings, similarity coefficients are employed to contrast the feature spectra in passed and failed executions and pinpoint the suspicious features.

In this paper, we have proposed a program feature known as Key Block Chains (KBCs), a suspiciousness estimation formula known as Minus, and a technique named MKBC. We have conducted a controlled experiment on three medium-sized subject programs to evaluate our technique. The results show that all these three ideas are promising.

Future work may include the study of the localization of multiple faults in a program, as well as potential generalization of our noise reduction technique.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2009*), pages 88–99. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing* (*PRDC 2006*), pages 39–46. IEEE Computer Society Press, Los Alamitos, CA, 2006.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference: Practice And Research Techniques* (*TAICPART-MUTATION 2007*), pages 89–98. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[4] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 5–15. ACM Press, New York, NY, 2007.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pages 189–200. ACM Press, New York, NY, 2008.

[6] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 82–91. ACM Press, New York, NY, 2006.

[7] P. Cellier. Formal concept analysis applied to fault localization. In *Companion of the 30th International Conference on Software Engineering* (*ICSE Companion 2008*), pages 991–994. ACM Press, New York, NY, 2008.

[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 34–44. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[9] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.

[10] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

[11] C. Fu and B. G. Ryder. Exception-chain analysis: revealing exception handling architecture in Java server applications. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), pages 230–239. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[12] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pages 167–178. ACM Press, New York, NY, 2008.

[13] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*), pages 184–193. ACM Press, New York, NY, 2007.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2005*), pages 273–282. ACM Press, New York, NY, 2005.

[15] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 16–26. ACM Press, New York, NY, 2007.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2005*), pages 15–26. ACM Press, New York, NY, 2005.

[17] C. Liu, L. Fei, X. Yan, S. P. Midkiff, and J. Han. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32 (10): 831–848, 2006.

[18] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 30–39. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[19] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 56–66. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot: a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (*CASCON 1999*), page Article No. 13. IBM Press, 1999.

[21] R. Vallée-Rai and L. J. Hendren. Jimple: simplifying Java bytecode for analyses and transformations. Technical Report 1998-4. Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[22] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC 2007*), volume 1, pages 449–456. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[23] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 81–90. ACM Press, New York, NY, 2008.

[24] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 201–210. ACM Press, New York, NY, 2008.

[25] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS 1010*), pages 143–154. ACM Press, New York, NY, 2010.

[26] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC 2009/FSE-17), pages 43–52. ACM Press, New York, NY, 2009.

[27] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and X. Wang. Fault localization through evaluation sequences. *Journal of Systems and Software*, 83 (2): 174–187, 2010.