Enhancing Contexts for Automated Debugging Techniques

Yan Lei, Chengsong Wang, Xiaoguang Mao and Quanyuan Wu
School of Computer
National University of Defense Technology
Changsha, China
yanlei@nudt.edu.cn, jameschen186@gmail.com, xgmao@nudt.edu.cn, quanyuanwu@nudt.edu.cn

***Abstract*—Most existing automated debugging techniques just focus on selecting a set of suspicious statements that may cause failures and ranking them in terms of suspiciousness. Therefore, these techniques always ignore the contextual information of how suspicious statements behave and propagate in the program. However, the contextual information is useful for discovering and understanding bugs. Hence, this paper proposes a novel approach to enhance contexts for automated debugging techniques. Based on localization results obtained from automated debugging techniques, our approach utilizes program slicing to classify suspicious statements into different contexts, and assigns different suspiciousness to the contexts and their elements. The experimental study shows that our approach can substantially improve debugging effectiveness.**

Keywords-automate debugging; program spectra; program slicing; statistical analysis.

I. INTRODUCTION

Software debugging has been recognized as one of the most time-consuming tasks in the development and maintenance of software [5]. With the aim at reducing the cost of debugging, a great number of research techniques are proposed to support automating or semi-automating the process of debugging and improve its performance [3-25].

However, existing automated debugging techniques just focus on selecting a subset of statements potentially responsible for failures and ranking them according to some criterion. Therefore, they ignore the contextual information of how suspicious statements behave and propagate in the program. The recent research [1] has found that the lack of contextual information may affect the activity of discovering and understanding bugs. For example, the faulty statement is included in the set of suspicious statements, and developers inspect the statements in this set. Due to the lack of contextual information, developers may judge the faulty statement is not responsible for program failures and just ignore this statement when inspecting it. Hence, it is vital to enhance the contexts for automated debugging techniques.

Program slicing technique [14-16] utilizes data and control dependence to identify the set of program statements that may affect or be affected by the values at some statement of a program. The set of statements is referred to as a program slice. A program slice can be essentially regarded as a context that shows a causal chain of how data and control propagates in a program. However, program slicing treats the statements of a program slice with the same suspiciousness to be faulty and thus lacks the guidance as to how the statements in a slice should be examined. That always leads developers to be frustrated and tiresome as the size of a slice can substantially increase with the increasing size and complexity of today's software [1]. One possible way to address this issue is to use the promising ability of some automated debugging techniques to assign different suspiciousness to the statements of a slice. More importantly, it implies that automated debugging techniques can adopt program slicing to enhance contexts for themselves.

Among current research, one promising automated debugging technique exploits the correlations between program entities and program failures via statistically analyzing coverage information [3-13]. This technique is generally referred to as spectrum-based fault localization (SFL). SFL usually collects coverage information and test results from dynamic executions to construct program spectra from passed and failed executions. After gathering spectra information, SFL adopts a ranking metric to evaluate the suspiciousness of program entities to be faulty and gives a ranking list of all entities in terms of suspiciousness. The research [3-13] has shown that SFL has a promising structure of evaluating the suspiciousness of an entity to be faulty. Nevertheless, SFL just outputs a ranking list of isolated entities and fails to provide the contextual information for discovering and understanding these suspicious statements.

Considering the popularity and ability, this paper chooses SFL and enhances contexts for it by using program slicing technique. Hence, we propose a debugging approach which uses program slicing to enhance contexts for SFL. Our approach utilizes SFL to compute the suspiciousness of each statement to be faulty. Then, the approach uses program slicing to identify the most suspicious statement and its relevant statements as a context showing how the most suspicious statement behaves in a program. Finally, except for the statements of all constructed contexts, the most suspicious statement of the remaining statements and its relevant statements constitute a new context, and this step would be iterated until each statement is classified into a particular context. Because a context is constructed from the most suspicious statement in it, our approach assigns the suspiciousness of this statement to the context. For each statement, its suspiciousness keeps unchanged. In addition, our approach offers two modes to developers with different experiences, and utilizes visualization and program

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

dependence to further assist developers. It can be seen that our approach provides useful contexts and recommends examining guidance of all contexts and their elements in terms of suspiciousness.

This paper conducts an experimental study on two standard benchmarks: the *Siemens suite* and *Space* [28], and compares our approach with nine ranking metrics of SFL: Ochiai [3], Jaccard [4], Tarantula [5], Wong2 [6], Wong3 [6], Ample [7], CBI [8], Optimal and Optimal^{P} [9]. The results demonstrate that the proposed approach outperforms all nine ranking metrics of SFL.

The remainder of this paper is organized as follows. Section II introduces the background of SFL. Section III describes our approach. Section IV conducts an experimental study. Related work is introduced in Section V and conclusions are given in Section VI.

II. BACKGROUND OF SFL

$$\begin{array}{cc} & N\ statements \qquad\qquad errors \\ M\ spectra & \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \end{array}$$

Figure 1. Input to SFL.

Spectrum-based fault localization (SFL) [3] is a dynamic program analysis technique. It typically uses coverage information of passed and failed runs to rank program entities whose activity correlates most with the failures. Passed runs are executions of a program that output as expected, whereas failed runs are executions of a program that output as unexpected. There are various types of entities, such as blocks, functions, branches, paths, etc. This study adopts statements as the entities.

First, we assume that a program P comprises a set of statements S and runs against a set of test cases T that contains at least one failed test case, with $N=|S|$ and $M=|T|$, respectively (see Fig. 1). The above matrix $M\times(N+1)$ represents the input to SFL. An element x_{ij} is equal to 1 if statement j is covered by the execution of test run i, and 0 otherwise. The error vector e at the rightmost column of the matrix represents the test results. The element e_i is equal to 1 if run i failed, and 0 otherwise. Except the error vector, the rest of the matrix is expressed in terms of matrix A. The ith row of A indicates whether a statement was covered by run i. The jth column of A indicates statement j was covered by which runs.

SFL usually measures the suspiciousness of a statement to be faulty from similarity between its statement spectra and error vector in the above matrix (see Fig. 1), and finally outputs a ranking list of all statements in descending order of suspiciousness. The similarity is quantified by ranking metrics.

TABLE I shows nine ranking metrics of SFL and how the suspiciousness of statement j was computed by the corresponding ranking metrics. $a_{pq}(j)=|\{i|x_{ij}=p\wedge e_i=q\}|$, and $p, q\in\{0,1\}$. $a_{0q}(j)$ represents the number of passed (q=0) or failed (q=1) test cases that do not execute statement j. $a_{1q}(j)$ denotes the number of passed (q=0) or failed (q=1) test cases that execute statement j.

TABLE I. FORMULAS OF SFL

Name	Formula	Name	Formula
Ochiai	$\frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j))*(a_{11}(j)+a_{10}(j))}}$	Jaccard	$\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)+a_{10}(j)}$
Tarantula	$\frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}+\frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}}$	Ample	$\left\lvert\frac{a_{11}(j)}{a_{01}(j)+a_{11}(j)}-\frac{a_{10}(j)}{a_{00}(j)+a_{10}(j)}\right\rvert$
CBI	$\frac{a_{11}(j)}{a_{11}(j)+a_{10}(j)}-\frac{a_{11}(j)+a_{01}(j)}{a_{11}(j)+a_{01}(j)+a_{00}(j)+a_{10}(j)}$	Wong2	$a_{11}(j)-a_{10}(j)$
Optimal	$\begin{cases}-1 & \text{if } a_{01}(j)>0 \\ a_{00}(j) & \text{if } a_{01}(j)\le 0\end{cases}$	Optimal^{P}	$a_{11}(j)-\frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)+1}$
Wong3	$a_{11}(j)-h$, where $h=\begin{cases}a_{10}(j) & \text{if } a_{10}(j)\le 2 \\ 2+0.1*(a_{10}(j)-2) & \text{if } 2<a_{10}(j)\le 10 \\ 2.8+0.001*(a_{10}(j)-10) & \text{if } a_{10}(j)>10\end{cases}$		

SFL is widely accepted and studied as a promising automated technique in the debugging community, and the research has empirically proven that SFL is effective to correlate faulty statements with failures in terms of suspiciousness [3-13]. According to the popularity and ability, this study chooses SFL to evaluate the suspiciousness of contexts and their elements.

III. THE APPROACH

Program slicing as a debugging aid was introduced by Mark Weiser [14]. Korel and Laski afterwards proposed dynamic slicing to focus on an execution in a specific input [15]. Because the localization result of SFL is based on the executions of a set of test cases instead of a specific execution, this study adopts static slices for our approach. There are two types of static slices: static backward slices (SBS) and static forward slices (SFS). The SBS of a variable at a statement includes all those statements which affect the value of the variable at that statement through chains of static data and/or control dependence [14]. In contrast, the SFS of a variable at a statement includes all those statements that are affected by the value of the variable at that statement through chains of static data and/or control dependence [19]. It can be seen that SBS can find a set of statements affecting a statement while SFS can identify a set of statements affected by a statement.

The basic idea of our approach is to apply program slicing to SFL by constructing different suspicious contexts and their elements for discovering and understanding faults. Our approach uses both SBS and SFS to construct a context showing how a statement affects and is affected by other statements in a program, and then utilizes SFL to evaluate the suspiciousness of each context and its elements.

The Algorithm 1 describes our approach. First, there are some explanations for Algorithm 1. This section adopts the program P and set of test cases T defined in Section II. The program P consists of a set of statements S that is denoted as $\{s_1, s_2, \ldots, s_N\}$. *FSlice*($s_i$) represents the union of the SFS of each variable at statement s_i. *BSlice*(s_i) denotes the union of SBS of each variable at statement s_i. *contextSet* stores all constructed contexts.

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

Algorithm 1 The proposed approach

```
1: Step 1: Compute the suspiciousness of each statement.
2:      SFLAnalyze(S,T);
3: Step 2: Construct different suspicious contexts and their statements.
4:      while(S is not empty){
5:        stmSet=GetMostSuspiciousStm(S);
6:        context=Φ;
7:        for(i=0; i<stmSet.size; i++)
8:          context=context∪BSlice(stmSet[i])∪FSlice(stmSet[i])∪stmSet[i];
9:        context.suspiciousness=stmSet[0].suspiciousness;
10:       S=S−context;
11:       if(mode=="weak-contexts")
12:         for(i=0;i<contextSet.size;i++)
13:           context=context−contextSet[i];
14:       Add(context,contextSet);
15:     }
16: Step 3: Output the localization results.
17:     if(visualization == true)
18:       Visualize(contextSet);
19:     else
20:       Text(contextSet);
```

Step 1: Compute the suspiciousness of each statement. The function *SFLAnalyze*(*S*,*T*) analyzes the statement coverage information and test results of test cases *T*, and adopts SFL to compute the suspiciousness of each statement in *S*. The format of statement coverage information and test results of test cases *T* is a matrix as shown in Fig. 1. In this study, for the statements with the same assigned suspiciousness, SFL ranks them in descending order of their line numbers in the source code. This strategy is also adopted by the *text-form* of localization results mentioned in the following step 3.

Step 2: Construct different suspicious contexts and their elements. This step iteratively constructs suspicious contexts until each statement is classified into a specific context. As the suspiciousness of some statements may be the same, the function *GetMostSuspiciousStm*(*S*) may return a set of most suspicious statements in *S* more than one statement. Lines 6 to 8 represent that besides the most suspicious statements, *context* consists of a set of statements that can affect or are affected by at least one of the most suspicious statements in *S*. As a context is constructed from the most suspicious statements in *S*, line 9 assigns the suspiciousness of the most suspicious statements to its corresponding context. Line 10 excludes the statements of the new context from the set of statements *S*.

There are two modes in our approach: *weak-contexts* and *strong-contexts*. The mode of *weak-contexts* demands that a statement can be only classified into one context. Therefore, it can cause some less suspicious contexts miss those statements presented in a more suspicious context. Lines 11 to 13 exclude those statements of all constructed contexts from the new context when the mode is *weak-contexts*. In contrast, the mode of *strong-contexts* maintains the integrity of each context. In this mode, a statement may be classified into different contexts. The step 3 presents a strategy to give some hints of the repetitive appearance of a statement in different contexts. Line 14 inserts the new context into *contextSet* that contains all constructed contexts.

Step 3: Output the localization results. This step outputs localization results in two different forms. One is *text-form* and the other is *visualization-form*. The functions *Text*(*contextSet*) and *Visualize*(*contextSet*) process the *text-form* and *visualization-form* of the localization results *contextSet* respectively. The *text-form* firstly ranks all contexts in descending order of their suspiciousness and then sequences the statements of each context in descending order of suspiciousness given by SFL. The ranking list is outputted in a text form. The *visualization-form* uses program dependence graphs [26,27] or lists the statements in a context along the program dependence edges from the starting point to show each context and its statements, and maps color to them according to the suspiciousness. This form can visually assist developers in understanding and locating faults. Some information is attached to each statement, such as *suspiciousness* and *role*. There are three types of *role*, namely "*root*", "*affect*" or "*affected*". The "*root*" denotes the statement is chosen to be sliced to construct the context. The "*affect*" and "*affected*" represents the statement affects or is affected by the "*root*" statement respectively. As mentioned in step 2, a context may be constructed from several statements with the same highest suspiciousness. Hence, a context may contain several "*root*" statements. In this case, the algorithm will number "*root*" statements to associate each "*root*" statement with its corresponding "*affect*" and "*affected*" statements.

The key idea of the color mapping algorithm is based on the visualization algorithm of Tarantula [5]. The color of a context or statement can be anywhere in the continuous spectrum of colors from red to yellow to green in descending order of suspiciousness. The contexts or statements are colored red to denote "danger" and indicate high likelihood of containing faults; those contexts or statements are specified green to denote "safety" and suggest little correlation with the failure; the contexts or statements are marked yellow to denote "caution" and imply a medium circumstance between "danger" and "safety". The visualization algorithm of Tarantula is implemented by *GIMP* and limits the suspiciousness to belong to [0, 1]. However, some metrics of SFL can produce a value of suspiciousness out of this range, such as Wong2, Wong3 and Optimal. In addition, this study chooses *GTK+* to implement the color mapping algorithm. Hence, a new color mapping equation is defined in Eq. (1).

$$color(s)=\begin{cases} Red=\begin{cases} Range, & \text{if } Rate \geq 0.5 \\ (\frac{Rate-0}{0.5})*Range=2*Rate*Range, & \text{if } Rate<0.5 \end{cases} \\ Green=\begin{cases} (\frac{1-Rate}{0.5})*Range=(2-2*Rate)*Range, & \text{if } Rate>0.5 \\ Range, & \text{if } Rate \leq 0.5 \end{cases} \\ Blue=0 \\ where,\ Rate=\frac{s.suspiciousness-minSuspiciousness}{maxSuspiciousness-minSuspiciousness} \text{ and } Range=65535 \end{cases} \quad .(1)$$

In Eq. (1), *s* represents a context or statement and *s.suspiciousness* denotes the suspiciousness of *s*. The *minSuspiciousness* and *maxSuspiciousness* are the minimum and maximum suspiciousness in all statements respectively. *GTK+* uses RGB model to produce different colors and specifies the values of each of the three basic colors at the range from 0 to 65535. In addition, the color mapping algorithm just needs red and green to generate the spectrum of colors for contexts and statements. As a result, *Range* and

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

Blue equal to 65535 and 0 in Eq. (1) respectively. It can be found that Eq. (1) can handle any range of the suspiciousness and map color to contexts and statements from red to yellow to green in descending order of suspiciousness. Note that if *maxSuspiciousness* equals to *minSuspiciousness*, all contexts and their statements will be colored green by the color mapping algorithm.

As mentioned in the step 2, although the mode of *strong-contexts* maintains the integrity of each context, checking those repetitive statements in different contexts may increase the workload of developers. A strategy is proposed to address this problem. Following the ranking list of all contexts in descending order of suspiciousness, this strategy examines the statements of each context in turn. When checking the statements of a context, some additional information is attached to those statements presented in previous examined contexts, such as *rank/total* and *pre-contexts*. The *rank/total* means the *rank* of the statement in *total* statements of the program in descending order of suspiciousness. The *pre-contexts* denote the set of higher ranked contexts that contains the statement before this context. The *rank/total* shows an indication of how suspicious a repetitive statement is in total statements of the program and the *pre-contexts* provides the connections of a repetitive statement in different contexts. In addition, the *visualization-form* colors gray to the repetitive statements. The above strategy offers some useful information of the repetitive statements and can alleviate the burden on developers for checking those repetitive statements.

As described above, it can be found that our approach can construct different suspicious contexts and their statements, and offer examining guidance of these contexts and statements in both *text-form* and *visualization-form*. It implies that the proposed approach equips SFL with contexts to further assist in discovering and understanding bugs.

IV. An Experimental Study

A. Experimental Setup

As SFL outputs a ranking list of all statements without repetition, this experiment use the *weak-contexts* mode of our approach to be compared to SFL. More concretely, the experiment study compares our approach in the *weak-contexts* mode with nine ranking metrics of SFL, namely Ochiai [3], Jaccard [4], Tarantula [5], Wong2 [6], Wong3 [6], Ample [7], CBI [8], Optimal and Optimal^P [9]. The formulas of these metrics are illustrated in TABLE I. This study chooses the *Siemens suite* and *Space* as the benchmarks because they are two widely used benchmarks in the field of software debugging with high quality. The two benchmarks can be obtained from the Software artifact Infrastructure Repository [28]. The *Siemens suite* contains 7 programs and 132 faulty versions of these programs. The *Space* contains 38 faulty versions. We select "*universe*" suite that contains all test cases in TABLE II. TABLE II lists the programs, the number of faulty versions of each program, lines of statements, lines of executable statements, number of all test cases, as well as the functional descriptions of the corresponding program.

TABLE II. Description of the *Siemens Suite* and *Space*

Program	Versions	LOC	Ex	Test	Description
print_tokens	7	563	203	4130	Lexical analyzer
print_tokens2	10	508	203	4115	Lexical analyzer
replace	32	563	289	5542	Pattern recognition
schedule	9	410	162	2650	Priority scheduler
schedule2	10	307	144	2710	Priority scheduler
tcas	41	173	67	1608	Altitude separation
tot_info	23	406	136	1052	Information measure
Space	38	9564	6218	13585	ADL interpreter

Although there are 170 versions in total, we were unable to adopt all of them. Because there was no failed test case in version 32 of *replace*, version 9 of *schedule2* and versions 1, 2, 34 of *Space*, we excluded the five versions. Additionally, we focus on executable statements, so the modifications of header files and definition/declaration errors were ignored. Hence, versions 4 and 6 of *print_tokens*, version 12 of *replace*, versions 13, 14, 36, 38 of *tcas* and versions 6, 10, 19, 21 of *tot_info* were also discarded. Finally, 154 faulty versions were used for the experiment.

In the experiment, the coverage information is gathered by using *Gcov* tool. We use FEMA (Failure Modes and Effects Analysis) slicing tool [26] developed by our group to perform program slicing. In addition, we adopt the *GTK+* to implement the algorithm of the visualization of our approach.

B. Evaluation metrics

The effectiveness of debugging techniques is widely evaluated by the percentage of code that needs to be examined (or not examined) to find the fault [6]. This evaluation assumes that developers will follow the ranking list to examine all statements from top to bottom until they encounter the faulty statement. Following this notion, we define *fault-localization accuracy* (referred as *Acc*) as the percentage of executable statements to be examined before finding the actual faulty statement [10]. A lower value of *Acc* indicates higher effectiveness.

For a more detailed comparison, we adopt *relative improvement* (referred as *Imp*) [10]. The *Imp* is to compare the total number of statements that need to be examined to find all faults using our approach versus the number that need to be examined by using the SFL. A lower value of *Imp* shows better improvement that our approach obtains.

C. Results and analysis

Figure 2 illustrates the *Acc* comparison between SFL and our approach in all faulty versions. The x-axis represents the percentage of executable statements to be examined. The y-axis denotes the percentage of faulty versions. A point in Fig. 2 represents when a percentage of executable statements is examined in each faulty version, the percentage of faulty versions has located their faults.

As shown in Fig. 2, the curves of our approach are usually higher than those of the corresponding metrics of SFL. It suggests that our approach improves the effectiveness of the nine metrics of SFL.

For a more detailed comparison, Fig. 3 presents the *Imp* of our approach over each metric of SFL in each program. The x-axis represents the name of each program. The y-axis

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

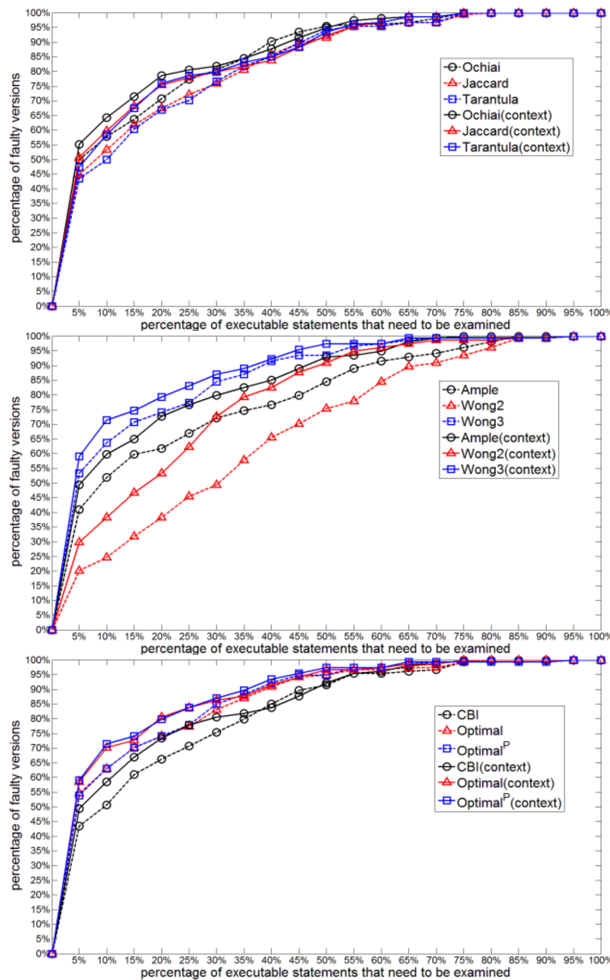
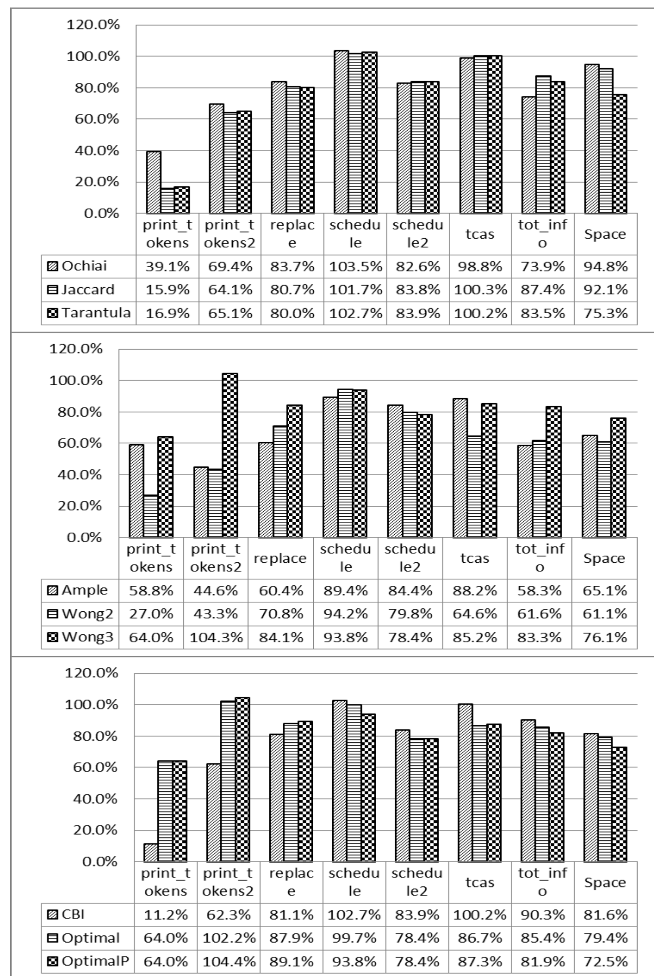


Figure 2. *Acc* comparison between SFL and our approach.

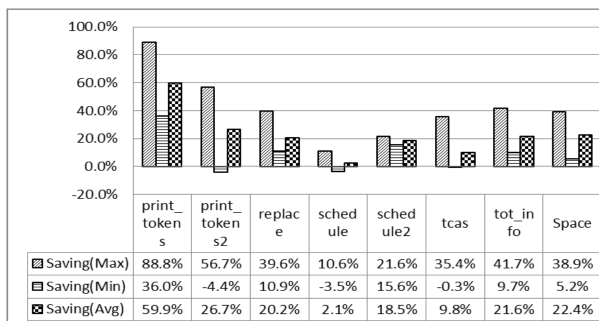


	print_tokens	print_tokens2	replace	schedule	schedule2	tcas	tot_info	Space
Ochiai	39.1%	69.4%	83.7%	103.5%	82.6%	98.8%	73.9%	94.8%
Jaccard	15.9%	64.1%	80.7%	101.7%	83.8%	100.3%	87.4%	92.1%
Tarantula	16.9%	65.1%	80.0%	102.7%	83.9%	100.2%	83.5%	75.3%

	print_tokens	print_tokens2	replace	schedule	schedule2	tcas	tot_info	Space
Ample	58.8%	44.6%	60.4%	89.4%	84.4%	88.2%	58.3%	65.1%
Wong2	27.0%	43.3%	70.8%	94.2%	79.8%	64.6%	61.6%	61.1%
Wong3	64.0%	104.3%	84.1%	93.8%	78.4%	85.2%	83.3%	76.1%

	print_tokens	print_tokens2	replace	schedule	schedule2	tcas	tot_info	Space
CBI	11.2%	62.3%	81.1%	102.7%	83.9%	100.2%	90.3%	81.6%
Optimal	64.0%	102.2%	87.9%	99.7%	78.4%	86.7%	85.4%	79.4%
OptimalP	64.0%	104.4%	89.1%	93.8%	78.4%	87.3%	81.9%	72.5%

Figure 3. *Imp* of our approach over each metric of SFL on each program.



	print_tokens	print_tokens2	replace	schedule	schedule2	tcas	tot_info	Space
Saving(Max)	88.8%	56.7%	39.6%	10.6%	21.6%	35.4%	41.7%	38.9%
Saving(Min)	36.0%	-4.4%	10.9%	-3.5%	15.6%	-0.3%	9.7%	5.2%
Saving(Avg)	59.9%	26.7%	20.2%	2.1%	18.5%	9.8%	21.6%	22.4%

Figure 4. The maximum, minimum and average saving of our approach over each program.

denotes the *Imp* in a specific metric of SFL. The tables in Fig. 3 show the detailed values of *Imp* on each program. If the value of *Imp* is less 100%, it means that our approach promotes the effectiveness of SFL. Otherwise it indicates our approach decreases the effectiveness of SFL.

As shown in Fig. 3, the values of *Imp* over each metric of SFL are less than 100% in most of programs. This indicates that the effectiveness of SFL is improved by our approach in most of programs. Take Ochiai as an example. The lowest *Imp* is 39.1% in *print_tokens*. This implies that our approach obtains the maximum improvement over Ochiai in *print_tokens*. It also means that when locating all faults in *print_tokens*, our approach only requires the examination of 39.1% of the number of statements that Ochiai requires the examination of. This represents a 60.9% saving in terms of effort, which is the maximum saving that our approach obtains in Ochiai. However, the highest *Imp* is 103.5% in *Schedule*, which implies that our approach requires an extra 3.5% effort to locate all faults in *Schedule* compared to Ochiai. This represents the minimum saving, -3.5%, that our approach obtains in Ochiai.

Fig. 4 illustrates the maximum, minimum and average saving of our approach in each program. As shown in Fig. 4, the average maximum saving that our approach obtains is 41.7% and the average minimum saving is 8.6%. On average, the saving of our approach is 22.7%, which indicates our approach is more effective than SFL.

Because the high suspicious statements evaluated by SFL are usually relevant to the faulty statements, our approach can classify the faulty statements into more suspicious contexts and they finally obtain higher ranks compared to those ranks in SFL. However, the high suspicious statements sometimes may be irrelevant to the faulty statements. Thus,

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

it can cause some less suspicious statements that are relevant to the high suspicious statements surpass the faulty statements in more suspicious contexts. This reveals the reason why our approach slightly decreases the effectiveness of some metrics of SFL in several programs.

We also found that the decrease of effectiveness, to some extent, is caused by the vulnerability of SFL. For example, SFL utilizes coverage information that cannot identify those statements whose execution affects the program output. Suppose that a non-faulty statement that is irrelevant to the faulty statement and has little contribution to the faulty output. Suppose further that the number of failed test runs executing the non-faulty statement is larger than that of failed test runs covering the faulty statement, whereas the number of passed test runs covering the non-faulty statement is less than that of passed test runs executing the faulty statement. In this case, SFL usually assigns higher suspiciousness to the non-faulty statement than that to the faulty statement. If a context is constructed from this non-faulty statement, it is more probable for some less suspicious statements associated with this non-faulty statement to surpass the faulty statement in this more suspicious context. However, if SFL can identify those statements whose execution affects the program output, SFL can rank this non-faulty statement lower than the faulty statement. Under this circumstance, it reduces the possibility of decreasing the effectiveness of SFL when using our approach.

D. *Threats to Validity*

A threat to the validity of our experiment is the subject programs used by the study. The experiment chooses the *Siemens suite* and *Space* because they are two de-facto benchmarks in the field of software debugging. Apparently, the results obtained may not apply to all programs. For instance, a program, in reality, usually ships with multiple faults rather than a single fault as used in our experiment. The recent research [2] has found that multiple faults pose a negligible effect on the effectiveness of fault localization, and even in the presence of many faults, at least one fault is found by the fault localization technique with high effectiveness. Although these findings increase our confidence in the effectiveness of our approach for locating multiple faults, they cannot guarantee that multiple faults create a negligible effect on the effectiveness of our approach. It is necessary to use more subject programs to further investigate the effectiveness of our approach.

Another threat is the metrics of SFL adopted by our experiments. The experimental study selects nine metrics of SFL to empirically evaluate the effectiveness and applicability of our approach. However, SFL is a big family and contains many metrics [3-13]. Our approach may not be applicable to some other metrics of SFL. It is vital to apply our approach to a much broader spectrum of SFL to further evaluate its effectiveness and applicability.

V. RELATED WORK

Spectrum-based fault localization (SFL) has motivated plenty of debugging techniques over recent years. The effectiveness of SFL highly depends on the ranking metrics that measure the correlations between program entities and failures. Hence, many metrics of SFL are proposed, such as the nine metrics of SFL adopted by the experiment [3-9]. In addition, there are many types of program entities presented for SFL, such as statements [5,6,9,10], blocks [3,7], branches [8], etc. Some new complex coverage types of program entities using dependences or flow are also proposed to strengthen the relationship among the elements of a program entity, such as mixed coverage [11], information flow coverage [12] and control flow edge coverage [13]. Although all of the above approaches have delivered the promising ability in correlating program entities and failures, they usually ignore the fact that the contextual information is useful for discovering and understanding the bugs. To enhance contexts for SFL, our approach applies program slicing to SFL to construct different suspicious contexts and their statements.

Program slicing technique [14-16] has also been widely studied in the field of debugging. Kusumoto et al. [17] conducted an experimental evaluation of program slicing for fault localization and Zhang et al. [18] investigate the effectiveness of dynamic slicing in locating faults. Their research shows program slicing is useful for fault localization. To further narrow down the searching scope, Gupta et al. [19] present failure inducing chops that intersect the forward dynamic slices of inputs with the backward dynamic slices of outputs. Zhang et al. [20] study the probable missing dependencies in dynamic slices and use an effective slicing approach to locate execution omission errors. Xin et al. [21] present a data-centric dynamic slicing technique that focuses on the dependencies in memory locations. Zhang et al. [22] propose an event-centric dynamic slicing technique that removes the irrelevant events from the sets of events to narrow down the searching scope of events. Although slicing-based debugging techniques have made great progress in these years, the size of a slice is still large. In addition, the elements of a slice are always treated with same suspiciousness to be faulty and no checking order is recommended to developers. Therefore, the slicing-based debugging techniques are rarely used in practice [2]. To alleviate this problem, our approach uses SFL to quantify the suspiciousness of a slice and its statements, and provides the guidance as to how the statements in a slice should be examined.

Baah et al. [23] uses the conditional probability in a dependence graph of a failed run to compute the suspiciousness of each node, and associate a state configuration with each node to construct a context and understand the problem. In contrast to their approach, our approach uses program slicing to iteratively construct different suspicious contexts and their statements according to the location results given by SFL.

Jiang et al. [24] proposes a context-aware statistical debugging approach by constructing and ranking the control-flow paths. The control-flow path is a context showing how a faulty predicate behave in a program. HOLMES [25] statistically analyzes path profiles of both passed runs and failed runs to isolate bugs that correlate with failure, and also uses paths to show a context where bugs occur. Unlike these

Copyright (c) The Government of China, 2012. Used by permission to IARIA. ISBN: 978-1-61208-230-1

two approaches, our approach uses program slicing constructs contexts and their elements, and utilizes SFL to assign suspiciousness to them. A context in our approach is essentially a slice showing how the most suspicious statement affects and is affected by other statements.

VI. Conclusion

This paper proposes a debugging approach to enhance contexts for a promising automated debugging technique, namely spectrum-based fault localization (SFL). The proposed approach applies program slicing to SFL by constructing different suspicious contexts and their elements for assist in understanding and locating faults. In addition, our approach offers two modes to different experienced developers, and uses the visualization and program dependence to further help understand the problem. The experimental study on two standard benchmarks shows that the proposed approach outperforms all nine metrics of SFL.

In future work, we plan to evaluate the effectiveness of our approach across a much broader spectrum of programs. We will also further study the applicability of our approach to more metrics of SFL and other automated debugging techniques.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China under Grant No.91118007, 90818024 and 61133001, the National High Technology Research and Development Program of China (863 program) under Grant No.2011AA010106 and 2012AA011201 and the Program for New Century Excellent Talents in University.

References

[1] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *the 2011 International Symposium on Software Testing and Analysis*, Toronto, Canada, 2011, pp. 199-209.

[2] N. DiGiuseppe and J. Jones, "On the influence of multiple faults on coverage-based fault localization," in *the 2011 International Symposium on Software Testing and Analysis*, Toronto, Canada, 2011.

[3] 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*, Windsor, UK, 2007, pp. 89-98.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *the 32nd International Conference on Dependable Systems and Networks*, Maryland, USA, 2002, pp. 595-604.

[5] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *the 24th International Conference on Software Engineering*, Orlando, USA, 2002, pp. 467-477.

[6] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *the 31st Annual International Computer Software and Applications Conference*, Beijing, China, 2007, pp. 449-456.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *the 12th Pacific Rim International Symposium on Dependable Computing*, Riverside, USA, 2006, pp. 39-46.

[8] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *the ACM SIGPLAN Conference on Programming Language Design and Implementation*, NY, USA, 2005, pp. 15-26.

[9] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology,* vol. 20, p. 11, 2011.

[10] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques," in *the 10th International Conference on Quality Software*, Zhangjiajie, China, 2010, pp. 13-22.

[11] R. Santelices and J. A. Jones, "Lightweight fault-localization using multiple coverage types," in *the 31st International Conference on Software Engineering*, Vancouver, Canada, 2009, pp. 56-66.

[12] W. Masri, "Fault localization based on information flow coverage," *Software Testing, Verification and Reliability,* vol. 20, pp. 121-147, 2010.

[13] Z. Zhang, W. Chan, T. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *the ESEC/FSE 2009,* Amsterdam, The Netherlands, 2009, pp. 43-52.

[14] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352-357, 1984.

[15] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters,* vol. 29, pp. 155-163, 1988.

[16] T. Gyimóthy, Á. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *the ESEC/FSE 1999*, Toulouse, France, 1999, pp. 303-321.

[17] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, pp. 49-76, 2002.

[18] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering,* vol. 12, pp. 143-160, 2007.

[19] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *the 20th International Conference on Automated Software Engineering*, Long Beach, USA, 2005, pp. 263-272.

[20] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," *ACM Sigplan Notices,* vol. 42, pp. 415-424, 2007.

[21] B. Xin and X. Zhang, "Memory slicing," in *the 18th International Symposium on Software Testing and Analysis*, Chicago, USA, 2009, pp. 165-176.

[22] X. Zhang, S. Tallam, and R. Gupta, "Dynamic slicing long running programs through execution fast forwarding," in *the 14th International Symposium on Foundations of Software Engineering*, Portland, USA, 2006, pp. 81-91.

[23] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering,* vol. 36, pp. 528-545, 2009.

[24] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *the 22nd International Conference on Automated Software Engineering*, Atlanta, Georgia, 2007, pp. 184-193.

[25] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *the 31st International Conference on Software Engineering*, Vancouver, Canada, 2009, pp. 34-44.

[26] W. Dong, J. Wang, C. Zhao, X. Zhang, and J. Tian, "Automating software FMEA via formal analysis of dependence relations," in *the 32nd Annual International Computer Software and Applications Conference*, Turku, Finland, 2008, pp. 490-491.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319-349, 1987.

[28] SIR, http://sir.unl.edu/portal/index.php

Copyright (c) The Government of China, 2012. Used by permission to IARIA. **ISBN: 978-1-61208-230-1**