*I.J. Information Engineering and Electronic Business*, 2013, 2, 8-14
Published Online August 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2013.02.02

# A Non-Parametric Statistical Debugging Technique with the Aid of Program Slicing (NPSS)

Farzaneh Zareie, Saeed Parsa
Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
Email: farzaneh_zareie@comp.iust.ac.ir, parsa@iust.ac.ir

***Abstract*—**A method is introduced in this paper, which promotes automated bug localization. It is based on the combination of two bug localization techniques, Non-Parametric Statistical Debugging and Backward Slicing. The proposed method, computes some vectors (called execution vectors) based on the status of each basic-block's execution in running of test-cases. According to the behavior of each basic-block in failed test-cases and passed ones, two likelihoods are computed and regards to them, basic-blocks become prioritized. At last static slice of program and dynamic backward slice for one failed test-case are computed. While seeking for faulty statement in ranked basic-blocks, the method either returns the basic-block's statements in the static backward slice or the part of it presented in the computed dynamic backward slice. NPSS has been applied on the Siemens test suite, space, grep and gzip. Our experimental study shows the accuracy and effectiveness of the method in accurate bug localization.

***Index Terms*—**Bug, Bug Localization, Program Backward Slicing, Statistical Debugging, Non-Parametric Statistical Relations.

## I. INTRODUCTION

Despite of all progresses in programming languages and test processes, there are bugs in software that bother its consumers and developers. Software debugging is one of the most important software lifecycle phases. It contains bug localization and bug correction [1]. After software delivery, costumers' feedback can help developers to find out which functionality encountered problems. They collect a number of positive or negative feedbacks [2]. Positive feedbacks play "passed test-cases" role and negative ones called "failed test-cases" in debugging process. Based on the fact that manual bug localization in software (especially in large ones) is a time-consuming process, using techniques to automate it, has become necessary. There are a lot of researches have been done in this area. Among them, three categories have been successful. These categories are Statistical Debugging [3][4][5][6][7], Program Slicing [2][8][9][10][11][12][13][14][15] and Delta Debugging [16][17][18]. All of them have their advantages and disadvantages. Statistical Debugging uses a large number of test-cases to study the behavior of statements. The difference between execution of a statement in failed test-cases and its execution in passed ones helps us to decide about its status of being faulty or non-faulty [3][4][5][6]. These methods use code instrumentation. Code instrumentation is the process of adding some extra code to the program to keep track of program execution. Interested points of instrumentation are usually predicators. Predicators contain conditional statements, return statements, loop's conditions and so on. After collecting data from test-cases, some statistical relations will be applied on them and statements become ranked [3][4][5][6]. It's obvious that all executed statements don't have influence on the output of program and we can ignore them while seeking for faulty statements.

Another technique which contributes to localize program bug, is Program Slicing. Program Slicing introduced by Weiser [8]. The main idea of program slicing was finding all possible statements that could have affect on a statement so it used static dependence graph. Obviously the statement which doesn't execute in a failed run could not have any influence on program failure. So the idea of Dynamic Program Slicing was emerged [9] that computes all executed statements that have influence on one statement. It uses dynamic dependence graph and results in a smaller set of statements. As the wrong output should have been affected by the faulty statement, for finding the faulty statement it's enough to compute dynamic backward slice for the wrong output where bug appears. However, it's not always true. Sometimes, the faulty statement (program bug) changes the path of execution so the wrong output doesn't have been affected by it. Forward and Relevant Slicing were introduced to overcome this problem [10][19][20]. Using just one failed test-case, dynamic program slicing generates a relatively large set of statements. So some methods have developed based on intersection, union and etc of program slices such as dicing [11], multiple point slicing [10], chopping [12][13] and other techniques introduced in [14][15][21] to reduce suspicious statements. Delta Debugging is the last category described in this paper. This method is based on finding the minimum set of failure inducing inputs. After finding this set, it computes forward dynamic slice for this set and the computed slice contains faulty

 

statement [16][17][18]. As this method is irrelevant to our work, we remain the subject for some who are interested in it. The technique proposed in this paper is based on Statistical Debugging and takes advantage of Program Slicing to reduce the number of suspicious statements.

In the introduced method, with the aid of some non-parametric statistical relations, the most important problem with statistical debugging techniques, needing a large number of failed and passed test-cases, has diminished. The other problem is that these methods can't find the origin of failure because they study predicates and therefore identify the area near the fault. So by applying program slicing this defect has eliminated. As said before, program slicing generates a large set of statements and there is no ranking method to prioritize its statements. However, two ranking methods have introduced on the subject [2][20], but neither of them can solve the problem efficiently and the problem with large set of statements still bothers debuggers. These problems inspired us to develop a new method which combines these two methods and prioritize the statements of slices using a statistical relation applied on them. The proposed method mainly contains four phases:

1. At first, the source code is instrumented.
2. Then the instrumented program executes with a limited number of test-cases (both failed and passed ones). The result of program execution saved into vectors. Each vector's element is the status of a basic-block's execution.
3. According to non-parametric relations introduced in our previous work [22], each element will become prioritized. After that, statements will be studied in order of their corresponding basic-blocks rank.
4. Dynamic Backward Slice for the output statement of a failed execution and Backward Static Slice for the program will be computed. While seeking the faulty statement, the method either returns the part of basic-block's statements presented in the static backward slice of output statement or just its statements presented in dynamic backward slice.

The subjects explained in this paper are organized as follows. In section 2 our motivations are mentioned, in section 3 and 4 we explain NPSS phases and the result of our experimental results on some middle and large size programs. In part 5 the conclusion and future studies are described.

## II. OUR MOTIVATIONS

As described in the previous section, our motivations are summarized as bellows:

1. As it's possible that there is not enough large set of test-cases, we were to find a method to help localize program bugs, using only few numbers of test-cases.
2. Using statistical debugging causes to find the area of the origin of the failure and therefore debugger had to search statements for finding the bug manually. It's clear that there are lots of statements irrelevant to program failure that should be considered. Using program slicing, this problem could be solved. So, we decided to combine them into a new method.
3. According to the fact that the slice size is large, we find it suitable to rank the slice statements according to their likelihood of being faulty.
4. Program slicing is a relatively time-consuming process. Therefore, computing backward slices for even a limited number of test-cases, takes long, especially for large programs. This method focuses on statistical debugging more than program slicing and computing dynamic backward slice for just one failed execution.
5. Focusing on statistical debugging rather than program slicing, helps to find more relevant bugs, too.

## III. NPSS METHOD

In this section, the steps of NPSS are described in details. Before describing how NPSS works, it's necessary to explain some basic definitions.

### *3.1. Basic Definitions*

**Basic-Block:** are maximal sequences of consecutive instructions with the properties that (1) the flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block and (2) Control will leave the block without halting or branching, except possibly at the last instruction in the block [23].

**Program Dependency Graph:** depicts the flow of information (contains data information and control information) among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed for computing the second [9][23]. Fig. 1 shows a piece of code and its Dependency Graph. Solid lines show control dependencies and dotted lines show data dependencies.

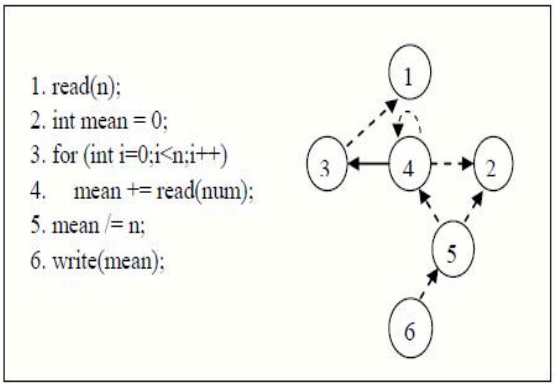


Figure 1. Dependency graph for a piece of code

**Program Static Backward Slicing:** is the process of

finding all the program statements that can affect a statement directly or indirectly [8]. To do so, Program Dependency Graph should be traversed backward from that statement we want to compute its slice.

**Program Dynamic Backward Slicing:** is the process of finding all program statements that certainly affect a statement directly or indirectly in a certain program running [9]. The dynamic backward slice for a statement in a certain run contains all statements in its static slice that have executed in that running.

### *3.2. How does NPSS work?*

As said in section 1, NPSS has four main phases. In this section each of these phases are addressed in details.

**Code Instrumentation and Computing Execution Vectors:** Regards to the fact that NPSS needs to keep track of basic-blocks execution in various running test-cases, we use an integer array with size of the number of program's basic-blocks. If the basic-block $b_i$ is executed in an especial test-case's running, $r_j$, the i-th element of $r_j$'s execution vector will be assigned to 1. Having the value equal to 0 for i-th element in a running execution vector, means that the basic-blocks $b_i$ didn't execute in that running. Fig. 2 shows an instrumented version of code has shown in Fig. 1 and computed execution vectors for three test-cases.

```
bb[0] = 1;
read(n);
int mean = 0;
for (int i=0; i<n; i++,bb[1]=1)
{
        bb[2]=1;
        mean += read(num);
}
bb[3] = 1;
mean /= n;
write(mean);
```

| input (n) | Computed Execution Vector |
|---|---|
| 0 | [1,0,0,1] |
| 1 | [1,1,1,1] |
| 2 | [1,1,1,1] |

Figure 2. An example of code instrumentation and computing execution vector

Sometimes the faulty code doesn't cause failure; rather it just causes an *Error* [1]. To make more accurate measurements, we have eliminated common execution vectors in failed and passed test-cases from passed set. If an execution vector is common in two sets, we assume that in passing run, an error has occurred that didn't affect the output. So we eliminate this execution vector from passed execution vectors' set.

**Ranking Basic-Blocks:** In this phase, we use a similar strategy in *Fuzzy-Slice* [22]. For the ranking of basic-blocks, two likelihoods of being faulty/ correct should be computed. As described in [22], we introduced two Likelihood of Being Faulty (LBF) and Likelihood of Being Correct (LBC) for each basic-block *b* as bellows:

$LBF_{b_i}$ is the amount of the contribution of basic-block $b_i$ to failures and is computed using (1).

$$LBF_{b_i} = \frac{\sum_{j=1}^{N} Execution - Vector_j.bb[i]}{N} \quad (1)$$

Consider N as the number of failed test-cases.

$LBC_{b_i}$ in a similar manner, is the amount of the contribution of basic-block $b_i$ to passed executions and is computed using (2).

$$LBC_{b_i} = \frac{\sum_{j=1}^{N} Execution - Vector_j.bb[i]}{N} \quad (2)$$

Consider N as the number of passed test-cases.

It's clear that how much a basic-block is executed in more failed test-cases and less passed test-cases, the likelihood of containing faulty statement increases. So (3) for each basic-block $b_i$ is hold:

$$Score_{b_i} \propto \frac{LBF_{b_i}}{LBC_{b_i}} \quad (3)$$

We can convert (3) to (4) using a constant multiplication:

$$Score_{b_i} = c \times \frac{LBF_{b_i}}{LBC_{b_i}} \quad (4)$$

Where c is a constant. In many test-cases, it has been seen that an assumed basic-block, $b_i$, is executed in a few number of failed test-cases and in none of passed test-cases. In such situations $Score_{b_i}$ is a very large amount while it shouldn't be. So we compute c as the difference between LBF and LBC. Regards to c, (4) is changed to (5).

$$Score_{b_i} = (LBF_{b_i} - LBC_{b_i}) \times \frac{LBF_{b_i}}{LBC_{b_i}} \quad (5)$$

$Score_{b_i}$ for all basic-blocks, is the metric for ranking them. How much $Score_{b_i}$ is greater, the likelihood of containing faulty statement is greater. So the output of this phase is a sorted list of basic-blocks according to their likelihood of capturing faulty statement.

**Program Slicing:** The result of previous phase is an ordered list of basic-blocks. We can study basic-blocks' statements in a backward manner to find program bug. But is it necessary? The answer is clearly, "NO". We can reduce the number of suspicious statements using *Program Slicing*. In order to do so, in this phase, we can

compute *Dynamic Backward Slice* for the wrong output for a failed execution. There are two possible situations:

1. Bug presents in dynamic backward slice of the wrong output.
2. Bug doesn't present in the slice because of the fact that bug causes to a wrong path and probably it didn't affect program output (relevant fault) [15][19][20]. In such cases, the whole statements of basic-block should be considered. Instead of considering the whole basic-block's statements, we use static backward slice of wrong output. So program's bug is captured with high probability in a smaller set.

In NPSS, we compute both of dynamic and static backward slices of wrong output. If dynamic backward slice is not empty for a basic-block, *bb*, NPSS returns statements of *bb* which have presented in computed dynamic slice and otherwise it returns the statements of *bb* have presented in computed static slice.

## IV. EXPERIMENTAL RESULTS

To measure the accuracy of NPSS in accurate bug localization, we use several programs and compare its results with some well-known bug localization methods in statistical bug localization.

Our selected programs are Siemens Suite programs taken from *Software Repository Infrastructure* (*SIR*) [23]. This suite has seven middle sized programs: *tcas, print-tokens, print_tokens2, replace, schedule, schedule2* and *tot_info*. To evaluate our work more accurate, we use three large size programs: gzip, grep and space. These programs are available at *SIR*, too. Table 1 shows their properties.

Table 1. Siemens suite programs' properties

| Program Name | Number of Faulty Versions | | LOC | Number of Test-cases | Program Description |
|---|---|---|---|---|---|
| tcas | 41 | | 138 | 1608 | altitude separation |
| print_tokens | 7 | | 402 | 4130 | lexical analyzer |
| print_tokens2 | 10 | | 483 | 4115 | lexical analyzer |
| replace | 32 | | 516 | 5542 | pattern substituter |
| schedule | 9 | | 299 | 2650 | priority scheduler |
| schedule2 | 10 | | 297 | 2710 | priority scheduler |
| tot_info | 23 | | 346 | 1052 | statistic computation |
| grep-2.2-v1 | v3 | 5 | 11826 | 199 | pattern matcher |
| | v6 | | | | |
| | v10 | | | | |
| | v11 | | | | |
| | v14 | | | | |
| gzip-1.2-v4 | v2 | 6 | 5166 | 214 | file compressor |
| | v4 | | | | |
| | v5 | | | | |
| | v14 | | | | |
| | v15 | | | | |
| | v16 | | | | |
| **space** | **38** | | **6199** | **158** | **ADL interpreter** |

In the rest of this section we briefly describe statistical bug localization techniques namely Cooperative Bug Isolation [6], Tarantula [3], Sober [4], Fault Localization through Evaluation Sequences [7]. We also describe *Fuzzy-Slice* [22] and then their comparisons with NPSS are illustrated. In our experiment results we didn't have excluded versions with empty fault matrixes and total number of bugs is considered.

### *4.1. Cooperative Bug Isolation (CBI)*

This method is based on two conditional probabilities introduced by Liblit [5]. These values are *Context (p)* and *Failure (p)* for each predicate, *p*, and

are computed as bellows:

$$Context(p) = P(program\ \ fails \mid p\ is\ evaluated) \quad (6)$$

$$Failure(p) = P(program\ \ fails \mid p\ is\ evaluated\ as\ True) \quad (7)$$

Predicates ranking is done according to the difference between these values. Comparison of NPSS and Cooperative Bug Isolation has shown in Fig. 3.

*4.2. Sober*

Cooperative Bug Isolation can't find program's bug when predicate $p$ is evaluated *True* in all of the executions. Sober [4] eliminated the problem by taking to account both of failing and passing executions. In Sober two distributions of f ($\theta$|*passing executions)* and f ($\theta$|*failing executions)* for each predicate $p$, is computed. In these distributions $\theta$ is head probability. It uses these distributions to evaluate bias of predicate $p$. The evaluation bias for $p$ considered as $\boldsymbol{n_f/n_t+n_f}$ that $n_f$ shows the number of times that $p$ is evaluated as false and $n_t$ shows the number of times that $p$ is observed. Two distributions are computed for failing and passing runs and the difference between them is considered as predicates' rank. Fig. 3 shows the comparison of NPSS with Sober.

*4.3. Tarantula*

Tarantula is another statistical bug localization method [3]. It ranks all the statements of program regards to relation 8.

$$score(s) \propto \frac{\%passed(s)}{\%passed(s) + \%failed(s)} \quad (8)$$

Notice to the fact that this time how much this score is higher, it's dedicated that the statement $s$ is healthier. *passed(s)* shows the percentage of passed executions and *failed(s)* shows the percentage of failed executions that s is executed in. Comparison of our work and tarantula is presented in Fig. 3, too.

*4.4. Fault Localization through Evaluation Sequences (DES)*

This method presented in [7] first converts each combinational condition to some simple conditions. So each simple condition is inspected lonely. Ranking simple conditions is like Sober [4]. It uses Sober's biases for each simple condition and the difference between the behaviors of it in failed and passed executions, reveals program fault. Comparison of NPSS and this method is presented in Fig. 3.

*4.5. Fuzzy-Slice*

Fuzzy-Slice, computes the backward dynamic slices of multiple failing and passing runs, assuming that few of them are available, and also introduces an execution vector space where each single run is considered as an execution point in that. Using Fuzzy C-means clustering in addition to a novel ranking and pruning technique, it prioritizes statements to help find the origin of failure [22]. Since in Fuzzy-Slice we excluded tot-info, comparison of Fuzzy-Slice and NPSS is shown in Fig. 4 according to the percentage of located fault rather than their number.

Table 2 illustrates the result of our experiments on large programs namely grep, gzip and space. In this table the percentage of code inspection to localize program fault (%), presented by *CI* and the number of inspected statements presented by *IS*. As said before, we didn't have excluded those versions with empty fault matrix from our investigation. In space program, five versions have empty fault matrixes.

In Table 2 we also present maximum, minimum and average number of inspected statements (*IS*s) for localizing program bug. As it is shown in this table, in the worst case, in space program, we just need to inspect only about 3 percentage of code to find the faulty statement. In our best performance, in grep program, the amount of IS is less than 0.01 percentage of code (0.0084%). In average, NPSS can localizebug in about 0.5 percentage of code.

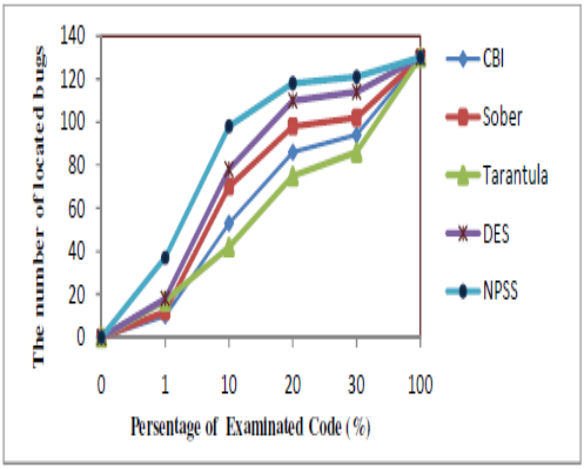


Figure 3. An Comparison of NPSS and Cooperative Bug Isolation (CBI), Sober, Tarantula and DES on Siemens Suite

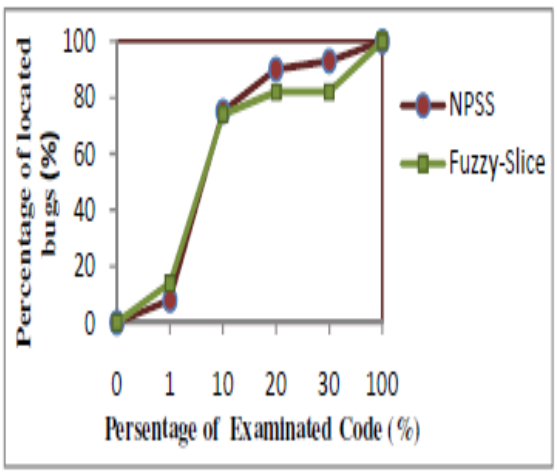


Figure 4. Comparison of NPSS and Fuzzy-Slice on Siemens Suite

 

Table 2. Result of NPSS on grep, gzip and space

| Program | Number of Faulty Versions | Max IS (% of LOC) | Min IS (% of LOC) | Mean IS (% of LOC) | #Located Faults with CI<1% | #Located Faults with CI<10% |
|---|---|---|---|---|---|---|
| grep-2.2-v1 | 5 | 109 (0.9%) | 1 (0.0084) | 26 (0.21%) | 5 | 5 |
| gzip-1.2-v4 | 6 | 141 (2.7%) | 7 (0.13%) | 30 (0.57%) | 5 | 6 |
| space | 38 | 200 (3.2% ) | 1 (0.016%) | 32 (0.51%) | 26 | 31 |

## V. CONCLUSIONS AND FUTURE WORKS

In this paper, we introduced a new method called NPSS. It combines statistical debugging and program slicing to localize program bugs. Considering basic-blocks, even large programs produce small execution vectors. NPSS applies non-parametric statistical relations on each basic-block to compute the likelihood of containing faulty/correct statements. After ranking basic-blocks, static backward slice of program and also dynamic backward slice of one failed test-case are computed. Statements of higher priority basic-blocks, should be inspected sooner. Instead of inspecting the whole statements, NPSS returns basic-blocks statements that are presented in dynamic backward slice if there is any and otherwise it assumes that bug causes relevant error and doesn't appear in dynamic slice and so returns it's statements presented in static backward slice. Applying NPSS on Siemens test suite, space, grep and gzip shows that it outperforms the other methods. We want to categorize statements according to their correlation and prioritize these categories instead of basic-blocks for future studies and also reduce the number of categorized statements that should be inspected using some heuristic methods such as genetic algorithms.

## REFERENCES


[1] P. Ammann and J. Offutt, Introduction to Software Testing. New York: the United States of America by Cambridge University Press, 2008.

[2] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation Ottawa, Ontario, Canada: ACM, 2006; 169-180.

[3] A. J. 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 Long Beach, CA, USA: ACM, 2005; 273-282.

[4] C. Liu, X. Yan, L. Fei, J. Han, and P. S. Midkiff, "SOBER: statistical model-based bug localization," in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering Lisbon, Portugal: ACM, 2005; 286-295.

[5] S. Horwitz, B. Liblit, and M. Polishchuk, "Better Debugging via Output Tracing and Callstack-Sensitive Slicing," IEEE Trans. Softw. Eng., 2010; 36 (1): 7-19.

[6] B. Liblit, M. Naik, X. A. Zheng, A. Aiken, and I. M. Jordan, "Scalable statistical bug isolation," SIGPLAN Not., 2005; 40 (6): 15-26.

[7] D. Jeffrey, N. Gupta, and R. Gupta, "Effective and efficient localization of multiple faults using value replacement," in ICSM 2009. IEEE International Conference on Software Maintenance, 2009; 221-230.

[8] M. Weiser, "Program slicing," IEEE Transactions on Software Engineering (TSE), 1982; 10 (4): 352–357.

[9] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation White Plains, New York, United States: ACM, 1990; 246-256.

[10] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," Softw. Pract. Exper., 2007; 37 (9): 935-961.

[11] T. Y. Chen and Y. Y. Cheung, "Dynamic Program Dicing," in Proceedings of the Conference on Software Maintenance: IEEE Computer Society, 1993; 378-385.

[12] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering Long Beach, CA, USA: ACM, 2005.

[13] K. Jens, "Slicing, Chopping, and Path Conditions


 

with Barriers," Software Quality Control, 2004; 12 (4): 339-360.
[14] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in Proceedings of the 28th international conference on Software engineering Shanghai, China: ACM, 2006; 272-281.
[15] T. Gyimothy, A. Beszedes, and I. Forgacs, "An efficient relevant slicing method for debugging," in Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering Toulouse, France: Springer-Verlag, 1999; 303-321.
[16] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," SIGSOFT Softw. Eng. Notes, 2000; 25 (5): 135-145.
[17] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" SIGSOFT Softw. Eng. Notes, 1999; 24 (6): 253-267.
[18] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," IEEE Trans. Softw. Eng., 2002; 28 (2): 183-200.
[19] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in Proceedings of the sixth international symposium on automated analysis-driven debugging Monterey, California, USA: ACM, 2005; 33-42.
[20] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," Empirical Softw. Eng., 2007; 12 (2): 143-160.
[21] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in Proceedings of the 25th International Conference on Software Engineering Portland, Oregon: IEEE Computer Society, 2003; 319-329.
[22] S. Parsa, F. Zareie, and M. Vahidi-Asl, "Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure," in Proceedings of the 10th international conference on Experimental algorithms, Crete, Greece, 2011; 352-363.
[23] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: principles, techniques, and tools, 2nd ed.: Addison-Wesley, 1986.
[24] Do Hyunsook, Elbaum Sebastian, and R. Gregg. "Software-artifact Infrastructure Repository", http://sir.unl.edu/portal/index.html [June 2012].

**Farzaneh Zareie** received her MS degree from Iran University of Science and Technology (IUST) under supervision of Prof. Saeed Parsa in 2011. She was a member of software testing research group in Prof. Parsa's Parallel Processing lab science 2009 and her research field is accurate bug localization techniques using program slicing. To that end, she has developed three non-parametric bug localization methods.

**Prof. Saeed Parsa** is one the academic staffs in Iran University of Science and Technology (IUST). He is an associated professor and received his PHD degree from the University of Salford in 1993. He was chairman of software group in the computer engineering faculty in IUST several years. He has supervised several research projects in Grid Computing, Software Testing and Parallel Processing and has published some books, many conference and journal papers in these areas. He has taught several PHD, MS and BS courses such as Super Compilers, Software Architecture, Formal Methods, Parallel Algorithms, Advanced Software Engineering and etc.

 