Factorising the Multiple Fault Localization Problem

Adapting single-fault localizer to multi-fault programs

Cheng Gong, Zheng Zheng
School of Automation Science and Electrical Engineering
Beihang University
Beijing 100191, China
rmgc606@126.com, zhengz@buaa.edu.cn

Yunqian Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zhangyq814@gmail.com

Zhenyu Zhang †
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zhangzy@ios.ac.cn

Yunzhi Xue
Laboratory for Internet Software Technologies
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
yunzhi@iscas.ac.cn

***Abstract*—Software failures are not rare and fault localization is always an important but laborious activity. Since there is no guarantee that no more than one fault exists in a faulty program, the approach to locate all the faults is necessary. Spectrum-based fault localization techniques collect dynamic program spectra as well as test results of program runs, and estimate the extent of program elements being related to fault(s). A popular solution is to generate a ranked list of suspicious candidates, which are checked in order, stopping whenever a fault is found. Such *single-fault localizers* locate one fault in one checking round, terminate, and wait to be triggered by the regression testing to validate the fixing of the located fault. In this paper, we study the manifestation of multiple faults in a program and propose an effective mechanism to indicate their presence. When a fault is reached during the checking round, we use it to interpret the failures observed, and update the indicator to judge whether there remain other faults in the program. Our indicator serves as a stopping criterion of checking the ranked list of suspicious candidates. Our work factorises the multiple fault localization problem into *developing single-fault localizers* and *adapting them to multi-fault programs*. It both improves the fault localization efficiencies of single-fault localizers, and avoids the ineffective efforts of thoroughly abandoning the many single-fault localizers to develop multi-fault localizers.**

***Index Terms*—fault localization, program spectra, multi-fault**

I. Introduction

With the rapid development of software systems, the importance of software quality and reliability has more and more been realized. By the influence of manual coding and increasing program scales, the release version of software is rarely found bug-free. Software testing is an effective method to show the presence of software faults. When a failure is observed during a program run over a test case, we know faults exist in the program, and accordingly mark a *failed* program run and a *failed* test case. However, even the existence of faults has been proved, not all the program runs will fail. It is because that a program run, which exercising the fault, may reveal no failure [12]. It increases the difficulty of fault localization.

† Corresponding author.

A. Background

Spectrum-based fault localization is a big family of automatic fault localization techniques. They collect dynamic program spectrum data, contrast their differences between the passes and failed communities (respectively collected from the passed and failed program runs), and estimate the location of faults by assessing the suspiciousness of program elements.

Tarantula [6] is a representative such technique. It calculates the ratio of passed runs and ratio of failed runs that exercising a program element, and estimates the suspiciousness of that program element being related to faults. Tarantula references thus calculated suspiciousness to sort the program elements in a descending mode, and outputs a ranked list of suspicious candidates (program elements). Previous studies showed that it is effective for a programmer to check along the ranked list of suspicious candidates to locate fault. Once a fault is reached during the checking, the checking round terminates and programmer stops to fix the fault, conducts regression testing to verify the fixing, and starts the next round of fault localization if any failed test case appears or remains [2][6][15]. This kind of techniques is referred to as *single-fault localizers* in this paper; and the above process is illustrated in Figure 1.

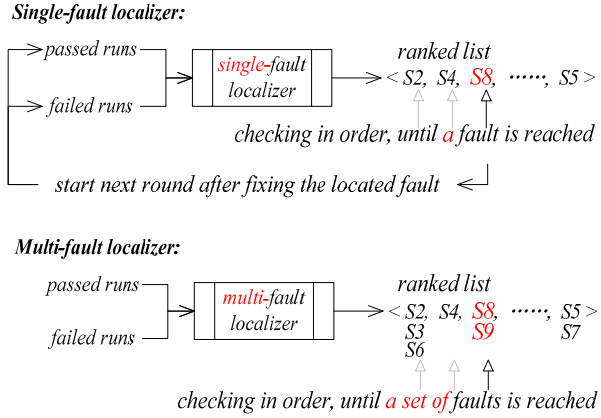

Figure 1. Different solutions to locate multiple faults

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.22

Based on the assumption that each failed run must be due to the exercising of at least one faulty program element, Steimann and Bertchler defined a fault localizer that "*takes all possible suspicious candidate combinations* ***explaining*** *a given set of failed runs into account*" [11]. Their work outputs a ranked list of suspicious candidate sets, which can be used to search for multiple faults simultaneously. We refer to this kind of techniques as *multi-fault localizers* (shown in Figure 1).

B. Problem and Drawbacks

First, *duplicated checking makes single-fault localizers ineffective*. Existing sing-fault localizers are designed to locate one fault in one checking round, stopping whenever a fault is reached. Having such a manner is because that blindly continuing to check for another fault (when a fault is located) can be very inefficient as there is no evidence of another fault. If the regression testing to validate the fixing of the located fault also generates failed test cases, another round of fault localization, as well as another round of checking, will be triggered. Let us consider the two testing process on the previous program v_1 and the revised program v_2 (due to fault fixing). For some program element, its dynamic spectra collected in the two testing process can be similar to each other since the program structure may not vary too much in the evolution from v_1 to v_2; and accordingly it may have similar suspiciousness values[1] in the two ranked lists l_1 and l_2. As a result, for some program elements, to a great extent, their orders in the two ranked lists l_1 and l_2 can resemble each other. To locate a fault in the second ranked list l_2, a programmer has to examine the program elements of higher rankings (i.e., prior to the faulty program element in l_2), but many of them may have been examined in the first ranked list l_1 during the first checking round. A heuristic is to skip the previously examined program elements when locating next faults using a single-fault localizer, but it can be very risky. In the motivation example in Figure 2, we will further demonstrate that.

Second, *the "debugging in parallel" solution relies on the accuracy of clustering*. Jones et al. [5] realized that it can be common for multiple faults to exist in one program, and propose to locate multiple faults in parallel. They cluster the failed runs according to dynamic spectrum information collected. By separately contrasting the program spectrum in each cluster of failed runs with the program spectrum in the set of passed runs, any single-fault localizer can be employed to locate multiple faults *in parallel* [5]. However, we realize that its effectiveness relies on the accuracy of clustering.

Third, *developing multi-fault localizers may not be the most cost-effective choice*. Single-fault localizers go through a long-term development and evolution. Many mature techniques have been proposed (e.g., [2][3][6][7][8][9][10][13][16]). They have different granularities [15], are based on different program elements [16], and work with different program languages [14]. Many optimization and enhancement have been made on them [3][9][12][16]. If we can adapt existing single-fault localizers to the use of locating multiple faults effectively (e.g., if we could locate more than one fault in one checking round), it is a great save of the efforts to design multi-fault localizers.

[1] For example, some program elements in the main function will be exercised in all the passed and failed runs, and always assigned a suspiciousness value of 0.5, by some fault localizers (e.g., Tarantula [6], Jaccard [2] or SBI [15]).

C. Our Ongoing Work

In this paper, we propose an indicator mechanism for single-fault localizers. When a fault is located during the checking of the resultant ranked list of suspicious candidates, we use it to interpret the observed failed runs, and query the indicator to judge whether there remain other faults in the program. It is used as a stopping criterion to decide whether to continue checking the rest candidates (for other faults).

The contribution of this ongoing work is at least two-fold. (i) The first work to adapt single-fault localizers to the use of locating more than one fault in one checking round. It improves the fault localization efficiencies of single-fault localizers. (ii) A universal indicator mechanism, which can be integrated with single-fault localizers that generate a ranked list of suspicious candidates. It factorises the multiple fault localization problem into two sub-problems, i.e., *developing single-fault localizers* and *adapting them to multi-fault programs*.

The rest of the paper is organized as following. Section II gives related work. Section III motivates our work. Section IV formulates our methodology. Section V concludes the papers.

II. Related Work

Naish et al. [9] summarized statement-level single-fault localizers and proposed optimal formulas under the assumption of existing no more than one fault in a program. In our work, we do not hold such a strong assumption.

Liblit et al. [7] proposed to install predicates into programs and manage to locate fault-relevant predicates. Their technique is of high scalability, compared to the statement-level fault localizer. They further proposed HOLMES [3] to locate fault in path-level. Liu et al. [8] proposed SOBER to locate fault-relevant predicates with the use of *evaluation bias*. Our work can be applied on predicate-based localizers if all failed run can be explained as exercising some fault-relevant predicate(s).

DiGiuseppe and Jones [4] conducted an experiment to evaluate the effectiveness of using single-fault localizer to locate faults iteratively in multi-fault programs. Their empirical results show that the effort to locate the first fault in multi-fault programs is comparable to that of locating fault in a single-fault program. Since the increasing in fault number will theoretically make it much probably to reach a fault, we foresee there is space for improvements and conduct this study.

Jones et al. [5] made use of clustering techniques to differentiate the failed runs due to different faults, and thus enables locating faults *in parallel* using single-fault localizers. We predict that their effectiveness is related to the accuracy of clustering. In the future work, we will compare with such kind of approaches (e.g., [17]) for effectiveness comparison.

Abreu et al. [1] used Bayesian reasoning to deduce multi-fault candidates and their probabilities, and proposed the technique BARINEL. Empirical study showed that BARINEL can outperform studied peer techniques with marginal higher complexity. In this paper, we do not involve probabilistic deducing. Instead of that, we deterministically indicate whether there exists more than one fault in a program.

Similar reasoning have been proposed in existing multi-fault localizers [1][11]. The insight of this work is to factorise the problem of locating multiple faults into *locating single-fault* and *adapt them to multi-fault programs*, and provide solution for the latter (since the former has been well studied).

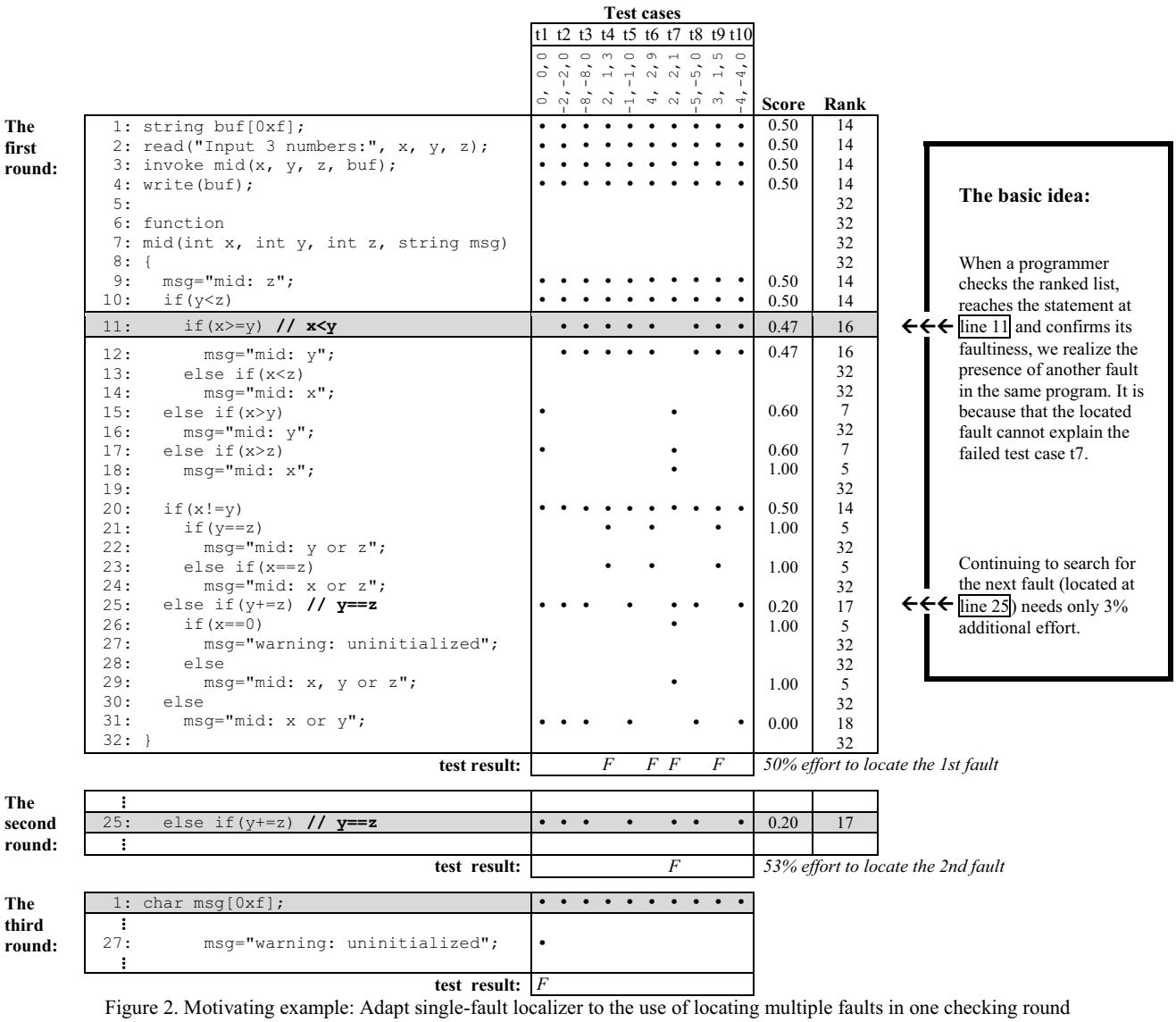

Figure 2. Motivating example: Adapt single-fault localizer to the use of locating multiple faults in one checking round

III. MOTIVATION

A. The Sample Multi-fault Program

Figure 2 shows a program excerpt[2] to find the middle number among three inputs. In this program, the main procedure (line 1-4) reads three inputs x, y, and z, invokes the function `mid` to find the median among them, and creates a buffer string `msg` to receive the answer. In the function `mid` (line 6-33), the values of `x`, `y`, and `z` are compared, and an answer is written into the buffer string. When all three inputs are zero, it is deemed uninitialized and will gives warning.

There exist at least two faults (line 11 and line 25) in the program. We choose ten test cases to run the program and find that four of them (t4, t6, t7, and t9) fail to output correct results. We use an "*F*" in the "**test result**" row to indicate a failed run.

[2] This example is taken from a previous study [15]. We expend it to demonstrate the presence of multiple faults in one program.

B. The Previous Approach

The failed test cases confirm the existence of fault(s) in the program. To drive a single-fault localizer to locate the fault(s), we capture the coverage status for each statement, in the program run with respect to each test case. In Figure 2, we use a dot "•" to indicate that a statement is exercised in a program run; otherwise, the cell is left blank.

Suppose a programmer is using Tarantula [6] to locate faults. As shown in Figure 2, statements at line 18, 21, 23, 26, and 29 are assigned the highest suspiciousness value and are given the highest ranking. They are examined first, but no fault exists in them. Statements 15 and 17 are examined next, and statements 1, 2, 3, 4, 9, 10, and 20 are examined after that. When statements 11 and 12 are examined, a fault is found at statement 11. Now 16 statements have been examined to locate the first fault, and the effort is 16/32=50%. The programmer stops checking the rest statements, fixes the fault in statement 11, conducts regression test on all the test cases, and finds a

failed run (t7) in the second round (as in Figure 2). This time, 53% effort (17 among the 32 statements are examined) is needed to locate the second fault (in statement 25).

C. Inspiring Our Work

By using Tarantula to locate the two faults (at statement 11 and 25), we examined 33 statements in total, in which many of them have been checked twice. An interesting phenomenon is that the ranking order of statements in the ranked list of the second round resembles that in the first round to a great extent. Can we simply skip the statements examined in the first round, when locating the second fault (in the second round)? The answer is ***No***, and in fact, such a heuristics is very risky. For example, after fixing the second fault, the regression testing is conducted on all the test cases and one run (t1) fails in the third round (shown in Figure 2). The third fault is found on statement 1, which unsuitably allocates space for a buffer string used in statement 27. However, since statement 27 is never exercised in the former two rounds, there is no clue to point out the faultiness of statement 1, even if it has been checked in the former two rounds. Blindly skipping the statements examined in previous rounds will have a high chance to miss a fault.

Let us revisit the checking round to search for clues. When the first fault (at statement 11) is located, we find that it is exercised in the program runs of test cases t2, t3, t4, t5, t6, t8, t9, and t10. We realize that it cannot *explain* [11] the failure observed with test case t7 because the located fault (statement 11) is not exercised in the program run of t7. As a result, we have evidence that there remain at least one fault in the program and the most promising action is to continue checking the rest candidates. The next suspicious candidate is statement 25, it is found faulty. As a result, we use 53% effort (17 among the 32 statements are examined) to locate two faults with one checking round. The improved efficiency is satisfactory. On the other hand, we notice that statement 25 is exercised in the program run of t7. Since the found faults (at statement 11 and 25) can explain all the observed failures, we stop checking, fix the two faults, and conduct regression testing as usual.

IV. Our Fault Indicator

We have demonstrated that using the located fault(s) to explain the observed failures provides clues on the existence of multiple faults in the program.

We now formulate our methodology as follows. Let $P = \{s_1, s_2, \ldots, s_n\}$ be the statements of a program P. $R = \{r_1, r_2, \ldots, r_m\}$ is the set of *failed* runs. We use $E_{i,j} = 1$ to denote that the statement s_i is exercised in the failed run r_j; $E_{i,j} = 0$, otherwise. $L_T = \langle s_{i1}, s_{i2}, \ldots, s_{in} \rangle$ is the ranked list of suspicious candidates generated by a single-fault localizer T.

Suppose the prior $t1$ candidates in T has been examined and s_{it1} is found to be a fault, the conventional solution is to stop checking the rest $n - t1$ candidates, fix s_{it1}, and conduct regression testing. Instead of that, we use an indicator I to judge whether there remain other faults (besides s_{it1}) in the program. **When *I* return "*yes*", we continue to check the rest candidates; otherwise, the checking is terminated.** We iterative this process and use $\{s_{it1}, s_{it2}, \ldots\}$ to denote the located faulty program elements. The indicator I is given as follows.

$$I = \begin{cases} yes & \text{if } \exists r_j, \forall s_{itk} \text{ s.t. } E_{itk,j} = 0 \\ no & \text{otherwise} \end{cases}$$

It is intuitively explained as follows. If there exists a failed run, in which all the located faulty statements are not exercised, it must be due to triggering some other faults (exercising faulty statements rather than the located ones). As a result, we have confidence in the existence of other fault in the program.

V. Conclusion and Extensions

To facilitate the laborious software debugging process, single-fault localizers and multi-fault localizers are proposed to automate the fault localization task. The former can be of lower efficiency, while abandoning the former and developing the latter family is not cost-effective. In this paper, we propose an indicator mechanism, which improves the fault-localization efficiency of single-fault localizers by providing a stopping criterion to enable them to locate more than one fault with one checking round. We thus factorise the fault localization problem into *developing single-fault localizers* and *adapt them to multi-fault programs*.

Extensions include model development and refinement. When a fault is located during the checking and the indicator confirms the existence of other faults, the suspiciousness of the remaining (not examined) candidates can be refined according to the located fault. We are also interested in the theoretical comparison with existing multi-fault localizers to know the benefit of the problem factorization approach in this paper.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, Spectrum-based multiple fault localization. In *ASE* 2009.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, A practical evaluation of spectrum-based fault localization. *JSS*, 2009.

[3] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani, HOLMES: effective statistical debugging via efficient path profiling. In *ICSE* 2009.

[4] N. DiGiuseppe and J. A. Jones, On the influence of multiple faults on coverage-based fault localization. In *ISSTA* 2011.

[5] J. A. Jones, J. F. Bowring, and M. J. Harrold, Debugging in parallel. In *ISSTA* 2007.

[6] J. A. Jones and M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE* 2005.

[7] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, Scalable statistical bug isolation. In *PLDI* 2005.

[8] C. Liu, L. Fei, X. Yan, S.P. Midkiff, and J. Han, Statistical debugging: a hypothesis testing-based approach. *TSE,* 2006.

[9] L. Naish, H. J. Lee, and K. Ramamohanarao, A model for spectra-based software diagnosis. *TOSEM*, 2011.

[10] M. Renieris and S.P. Reiss, Fault localization with nearest neighbor queries. In *ASE* 2003.

[11] F. Steimann and M. Bertchler, A simple coverage-based locator for multiple faults. In *ICST* 2009.

[12] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *ICSE* 2009.

[13] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai, Effective fault localization using code coverage. In *COMPSAC* 2007.

[14] J. Xu, W. K. Chan, Z. Zhang, and T. H. Tse. A dynamic fault localization technique with noise reduction for Java programs. In *QSIC* 2011.

[15] Y. Yu, J.A. Jones, and M.J. Harrold, An empirical study of the effects of test-suite reduction on fault localization. In *ICSE* 2008.

[16] Z. Zhang, W.K. Chan, and T.H. Tse, Fault localization based only on failed runs. *IEEE Computer,* 2012.

[17] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, Statistical debugging: simultaneous identification of multiple bugs. In *ICML* 2006.