Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Fault localization using disparities of dynamic invariants

Xiaoyan Wang[a,b,c], Yongmei Liu[b,*]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
[b] Department of Computer Science, Sun Yat-sen University, Guangzhou, China
[c] Department of Information Management and Information System, Nanjing Audit University, Nanjing, China

A R T I C L E   I N F O

Article history:
Received 19 April 2015
Revised 8 September 2016
Accepted 12 September 2016
Available online 13 September 2016

Keywords:
Software debugging
Fault localization
Dynamic invariant
Program analysis

A B S T R A C T

Violations of dynamic invariants may offer useful clues for identifying faults in programs. Although techniques that use violations of dynamic invariants to detect anomalies have been developed, some of them are restrained by the high computational cost of invariant detecting, false positive filtering, and redundancy removing, and others can only discover a few specific types of faults under a complete monitoring environment. This paper presents a novel fault localization approach using disparities of dynamic invariants, named FDDI. To make more efficient use of invariant detecting tools, FDDI first selects highly suspect functions via spectrum-based fault localization techniques, and then applies invariant detecting tools to these functions one by one. For each suspect function, FDDI uses variables that are involved in dynamic invariants that do not simultaneously hold in a set of passed and a set of failed tests to do further analysis, which reduces the time cost in filtering false positives and redundant invariants. Finally, FDDI locates statements that are data-related to these variables. The experimental results show that FDDI is able to locate 75% of 360 common faults in utility programs when examining up to 10% of the executed code, while Naish2, Ochiai and Jaccard all locate around 53%.

1. Introduction

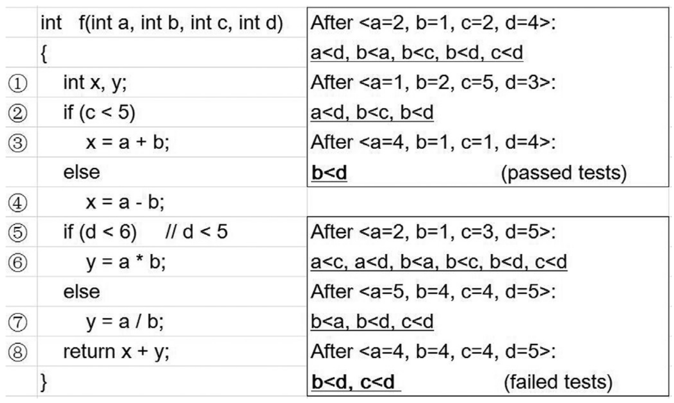
Dynamic invariants are relations among variables that are observed to hold at certain locations in some runs of a program (Nguyen et al., 2012). They can be inserted as assertion statements to detect abnormal behaviors of programs, collected to generate likely documentation and formal specifications, and used in program understanding (Zeller, 2009), etc. In particular, the violation of dynamic invariants can offer useful clues for fault localization and give better explanation for the localization result. However, current applications of dynamic invariants in automated program debugging are not efficient due to the high computational cost of detecting dynamic invariants. For Daikon (Ernst et al., 2001), it maintains an invariant pattern library that limits the detection of invariants with rich expressive power. GenInv (Nguyen et al., 2012) combines mathematical techniques to bring new capabilities to find more expressive dynamic invariants while increasing the complexity of the detection. For Diduce (Hangal and Lam, 2002) and ClearView (Perkins et al., 2009), they can only detect and patch specific types of errors due to the monitoring mechanisms they require. Currently, the work of Sahoo et al. (2013) combines dynamic program invariants with more sophisticated filtering techniques to identify a set of dynamic invariants that hold in selected passed runs but do not hold in failing tests, and returns the locations of the dynamic invariants as the localization results.

In this paper, we present a novel automated fault localization approach via using disparities of dynamic invariants, named *FDDI*, to locate the root causes of faulty programs via disparities of two sets of dynamic invariants generated from passing and failing test cases respectively. The intuition behind the idea is that a variable is likely to be related to the root cause if it is involved in relations that do not simultaneously hold in a certain number of failed runs and passed runs. In FDDI, spectrum-based fault localization (SBFL) techniques are first applied in function level to find suspect functions. Dynamic invariants are then yielded in these functions one by one. After detecting dynamic invariants, how does FDDI further locate the source code lines with bugs? In the following segment, we demonstrate how to use disparities of dynamic invariants to locate faults by using a snippet that contains a bug at line 5 where “if(d < 6)” should actually be “if(d < 5)”, as shown in Fig. 1. Consider the function `f()` in Fig. 1 and suppose we have only one invariant schema: i < j, where i and j are metavariables. At the entry of `f()`, instantiating this schema produces 12 concrete potential invariants:

```
a<b, a<c, a<d, b<a, b<c, b<d, c<a, c<b, c<d, d<a, d<b, d<c.
```

* Corresponding author.
E-mail addresses: wangxy25@mail2.sysu.edu.cn, xywang@nau.edu.cn (X. Wang), ymliu@mail.sysu.edu.cn (Y. Liu).

http://dx.doi.org/10.1016/j.jss.2016.09.014
0164-1212/© 2016 Elsevier Inc. All rights reserved.



	int f(int a, int b, int c, int d)	After <a=2, b=1, c=2, d=4>:
	{	a<d, b<a, b<c, b<d, c<d
①	int x, y;	After <a=1, b=2, c=5, d=3>:
②	if (c < 5)	a<d, b<c, b<d
③	x = a + b;	After <a=4, b=1, c=1, d=4>:
	else	**b<d** (passed tests)
④	x = a - b;	
⑤	if (d < 6) // d < 5	After <a=2, b=1, c=3, d=5>:
⑥	y = a * b;	a<c, a<d, b<a, b<c, b<d, c<d
	else	After <a=5, b=4, c=4, d=5>:
⑦	y = a / b;	b<a, b<d, c<d
⑧	return x + y;	After <a=4, b=4, c=4, d=5>:
	}	**b<d, c<d** (failed tests)

Fig. 1. Motivational example.

The upper right portion of Fig. 1 shows the set of invariants that have not been falsified by any of the preceding passed tests. Therefore, the last potential invariant "b < d" survived all three passed executions of function `f()`. Similarly, as shown in the bottom right portion of Fig. 1, the last potential invariant "b < d" and "c < d" survived all three failed executions of `f()`. As we can see, two likely invariant sets {b < d} and {b < d, c < d} are yielded via respectively running a passed test suite and a failed test suite, and the disparity between these two sets is {c < d}. FDDI extracts variable "c" and "d" from the disparity and can locate suspect statements line 2 and line 5 via using these variables. As a result, line 5 that is exactly the root cause is captured by our method.

FDDI is neither for certain error types nor under any monitoring environment. To be effective, it consists of two stages in its application. In the first stage, the block hit spectrum based technique is applied to rank functions by their suspiciousness, and n most suspicious functions will be selected for further analysis. By doing this, our method can concentrate on a small portion of the program at a time. Also, this overcomes the problem that a large number of variables in a program will bring existing invariant detecting techniques to their knees. In the second stage, for each suspicious function, two sets of dynamic invariants are first yielded by running a passed and a failed test case suite respectively. The statement-based reduction strategy (Yu et al., 2008) is applied to generate the passed and the failed test case suite. Variables in the difference of the two sets are then used to find data-related statements in the function by static analysis, which reduces the computational cost of filtering redundant and spurious invariants. These statements will be returned to developers in the order of the suspiciousness of the functions where they appear and in the order in which they appear in the same function.

The main contributions of this paper include: (1) We propose to use variables in the disparity of two dynamic invariant sets respectively generated from a failed and a passed test suite to locate bugs in a faulty program. (2) To reduce the high computational cost of current invariant detecting methods, FDDI employs existing dynamic invariant detecting tool, like Daikon, to generate dynamic invariants in the scope of one highly suspect function each time, and block hit spectrum based techniques are applied to rank these functions of a faulty program. (3) The experimental result shows that FDDI locates 75% of 360 common faults in 6 real-life utility programs when examining up to 10% of the executed code, while Naish2, Ochiai and Jaccard all locate around 53%.

The reminder of this paper is organized as follows: Section 2 investigates the proposed approach in detail. Section 3 evaluates the proposed approach and presents the experiment results. Section 4 presents the related work. Section 5 concludes this paper and highlights some future work.

2. FDDI

In this section, we first illustrate the top-level view of FDDI and present primary parts of FDDI. We then describe and analyze the algorithm of FDDI.

2.1. Top level view of FDDI

Fig. 2 depicts the overview of FDDI. The inputs of FDDI include a faulty program *app*, a test case suite *TS* and the component granularity *FL*. The output is a debugging report *R*. As we can see from Fig. 2, FDDI has eight primary components.

- *call DCC*: call DCC to calculate the suspiciousness of each executed function in *app*. DCC (Perez et al., 2014) is a dynamic coverage based multiple granularity fault localization technique. To employ DCC to rank functions, we need to set both the initial granularity and finial granularity to be the function level. As a result, a function sequence $\vec{F}$ is generated from this part.
- *select function*: select a function *f* from $\vec{F}$ in decreasing order of suspiciousness. FDDI will give deeper analysis of *f* via its dynamic invariants. FDDI proceeds to "generate report" part, if all functions with expected suspiciousness are selected out from $\vec{F}$.
- *refine tests*: find test cases that invoke *f* and remove redundant tests that cover the same code of *f*. FDDI distinguishes a test suite *RTS* that all test cases in it execute the code in *f* from the original test suite *TS*, because not all the tests in *TS* invoke the highly suspect function *f*. To save running time, FDDI removes redundant test cases that cover the same code of *f*, because these tests may not refine the generated dynamic invariant set of *f*. Besides, FDDI identifies a passed test suite RTS_p and a failed test suite RTS_f in *RTS*.
- *analyze call site*: analyze call sites in function *f*, and find the locations *CS* of loop and return statements and variable set *V* for each block in *f*. Instrumenting new call sites in *f* is required to obtain more dynamic invariants, because Daikon is applied in FDDI and it only generates dynamic invariants in the entrance and exit sites of a function by default.
- *instrument f*: instrument a dummy procedure for each executed loop in *f*, and one call site of the dummy procedure is instrumented in the loop head and another call site is added in the loop tail. Parameters of the dummy procedure are variables that hold in the loop. For each return statement, a dummy procedure is constructed and its call site is before the return statement. Detailed description can be seen in Section 2.2.
- *detect dynamic invariant*: detect two dynamic invariant sets S and S' for function *f* through respectively running the passed test suite RTS_p and the failed test suite RTS_f in Daikon.
- *analyze disparity*: find different dynamic invariants between S and S', and extract variables involved in these dynamic invariants. If any, FDDI finds suspect statements in *f* that are data dependent with these variables. Otherwise, FDDI collects all executed statements in *f* in a random selected failed test of *app*. FDDI returns back to "select function" part for the next suspect function.
- *generate report*: generate a debugging report of *app*. It consists of suspect statements in each selected function and corresponding variables in disparities. All suspect statements are returned in the report not only in decreasing order of suspiciousness of their functions but also in increasing order of appearance in their functions. Moreover, these variables with disparities could assist users in understanding the bug when they observe localization results in each function.