An Interactive Debugging Framework with Heuristic Graph

Xiaolin Ju *† ‡, Xiang Chen *‡, Shujuan Jiang ‡§, Junyan Qian ‡
*School of Computer Science and Technology, Nantong University, China
†School of Computer Science and Technology, Nanjin University, China
‡Guanxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, China
§School of Computer Science and Technology, China University of Mining and Technology, China
{ju.xl, xchencs}@ntu.edu.cn, shjjiang@cumt.edu.cn, junyanq@gmail.com

***Abstract*—Many fault localization approaches such as spectrum based fault localization(SBFL) generated a fault localization report based on which the programmer identifies the faults from corrected program entities. Unfortunately, the bug identification is quite tedious due to lack of sufficient context information. Furthermore, in the process of continuous searching for errors, the programmer's previous judgment makes no contribution for improving the efficiency of fault locating report. In this paper, we proposed an interactive debugging framework on which the programmer can iteratively identify faults with visualized heuristic graph. We first analyze the type of running information that is helpful for programmer's fault identification. The heuristic information is collected by program instrument, and modeled by program slicing techniques. We organized this information in the DOT format for visualization. In addition, we proposed an algorithm which can dynamically adjust the fault locating report using forward and backward slicing techniques based on the feedback of programmers. Furthermore, we proposed an enhanced algorithm that can adjust the fault locating report based on more than one feedback from programmers. Since the goal of fault-locating is to identify and fix faults in a program, our framework is more practical in program debugging with visualizing information.**

***Index Terms*—Interactive debugging, fault localization, heuristic information modeling, software testing**

I. Introduction

The ultimate goal of fault localization is to help the developers to identify and fix the faults quickly and accurately. Most of the existing fault localization technology outputs a report by giving a suspicious candidate program entities set or a ranking list of all program entities according to the descending order of error-proneness[1], [2]. Unfortunately, the entities in the report is treated as an independent entity and the relationships between them are usually ignored[3]. They are even not consecutive statements in the program. During software development and maintenance process, it is often difficult to determine whether the program statements are faults when reviewing them in the fault localization report, without any context information[4].

A recent study on debugging practice of professional software developers reveals that modern fault localization techniques are not widely used in practice [5]. Beyond fault localization report, we need more heuristic information to assist the developer to make a decision that an examined program entity is buggy or not. For example, to provide the execution context (e.g, data structure [6], data- and control-flow [7]) of the current examined statement, as well as the statements that are dependent on.

In addition, the previous fault localization technology often provides a one-time positioning information, and can not dynamically adjust the positioning report with the programmers previous work when debugging. This strategy will affect the accuracy and efficiency of the applied fault localization technology. It has been found that a lot of software faults are always in the back end of the fault localization report list, which weakens the effectiveness of the existing localization technology in practice[8]. Existing research indicates that the programmer's feedback can be useful for identifying the rest faults[9]. Therefore, we introduce a feedback mechanism in the debugging process, namely interactive debugging. The suspiciousness of unexamined statements are adjusted dynamically with slicing techniques with respect to the current statement judgment made by the programmer.

Furthermore, the previous fault localization technology, given direct results as a list of statements ranked by fault proneness, does not provide the relationship between the statements in the list[10], [11], [12]. the programmer needs to examine the source program to understand the role of these statements in the program so as to identify whether it is faulty or not. In many cases, because the statements in the list are not continuous in the source program, the programmer usually need to analyze the statement before and after the relevant statements, artificially explore the relationship between the statements to locate these faults. These debugging methods increase the effort of the programmer and the likelihood of human mistakes at the same time. In addition, most exiting fault localization approaches assume that each fault in source file is one-line bug[13]. However, there are some bugs that cross multiple lines of code in real-word software. We need more information about the relationship between these lines to identify the bug, because the impacts of these lines may depend on each other. To this end, with the program analysis technology applied, the accuracy of the relevant information will be accurate and other intuitive diagrams to the programmer to help debugging more easily[14], [15], [16].

In this paper, we answered three research questions as follow:

978-1-5386-3687-9/17 $31.00 © 2017 IEEE
DOI 10.1109/SATE.2017.14

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

- **RQ 1**: What are the types of heuristics information for locating faults, and how can we collect and model this information?
- **RQ 2**: To help the programmer identify software errors, how to arrange and visualize the heuristic information in a pattern similar to the developer's reasoning?
- **RQ 3**: Based on the feedback of the developers during program debugging, how can we dynamically adjust the fault localization report to improve the efficiency and effectiveness of fault-locating?

In the followed sections, we will illustrate the detailed algorithms and strategies proposed in our approach. Section II describes the heuristic information modeling including the running behaviors of program and how visualize the heuristic information with understandable graph. Section III presents how to dynamically adjust the fault locating report with considering of the feedbacks from programmers during debugging. Section IV provides the full view of our approach. We discuss related studies in Section V. We finally conclude and mention future work in Section VI.

II. Tracing and Modeling Heuristic Information of Program Runs

In this section, we will answer the research questions **RQ1** and **RQ2**. First, we describe the collecting and modeling method of running information. Then, we discuss the analysis algorithm of extracting the bug-related information for identifying bug and visualizing the heuristic information to assist the programmer in debugging.

A. Tracing and modeling the program running

In order to assist the programmer to identify software faults when debugging, it is necessary to provide program information as much as possible with the program faulty statements. This information can be divided into two types of static information and dynamic information. The dynamic information that the programmer is concerned with is the program execution path, and the variable value change related to the fault. While the static information includes the error statement depends on other statement information.

For the collection of static information, our mainly work is to construct the target program control flow chart(CFG). Control flow diagram construction technology has been highly developed. Therefor, no more details about the relevant technology described here. The resulting control flow graph will be utilized for subsequent stage model analysis and visualization.

Dynamic information is mainly to guide the understanding of the behavior of the program. The following information will be tracked in our method: the path of the program execution. Since all the statements in any basic block of the program perform the same track, we use the basic block as part of the execution trace. In addition, the predicates is also recorded in the execution trace since the they determine which basic block is to be executed next. Moreover, predicates are part of the entities that are most prone to error.

During the software debugging process, the programmer usually only concerned with statements that are closely related to the location of the program's failure output, not all statements in the program. In view of this, we do not have to provide all the information without filtering. Instead, we only provide a small number of statements and predicates that are closely related to suspicious error statements. We employ the queue to scroll through the recent execution of the program's dynamic information for program information visualization during the interactive debugging phase. Figure 1 shows the data structure for describing the program execution information tracking queue in our method and the variable table used by the queue elements.

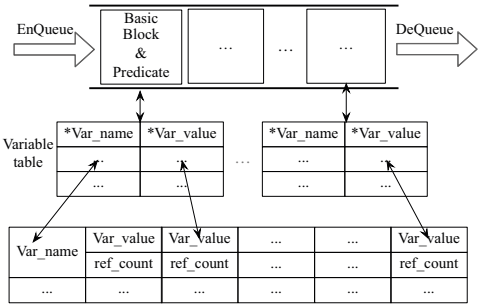

Fig. 1. Model of runtime traces

For Figure 1, we should note that: (1) The queue stored part of the execution sequence of basic blocks and predicates. We do not save all the follow-up sequence for two reasons: First, the program execution process may produce an infinite sequence of execution, which conflicts the storage space limits; Second, it is unnecessary to save all of the information at all. We save the information which is to assist the programmer to identify faults when debugging. Too much information on the programmer's judgment is not only helpless, but also harmful to the efficiency of judgments, and even interfere with the programmer to make the right judgments. (2) The variable table is to store the variables used by the base block or predicate in the queue. To increase processing speed and decrease storage space, our framework use the pointer to store the address of the actual storage location of the variable. Also the value of the variable is tracked by real-time recording. (3) In order to track the value of the variable history information, we save a certain period of time to change the value of the variable. Different basic blocks may use the same variable, but its value may change or remain unchanged when used. Therefore, we record the variable value (**Var_value**) and variable reference counter (**ref_count**). When the variable value changes, if the new value does not exist in the variable table, then we create a new value item and set the reference counter to 1. Otherwise, if the new value already exists in the table, the reference counter for the corresponding entry is incremented by 1. In practice, we apply a hash function mapping which can quickly locate and update the corresponding items in the table.

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

B. Model analyzing and visualizing

The goal of visualization is to present the programmer with heuristic information for identifying the current statement. Too much or too little information does not contribute to the programmer to make the right judgments. Following the principle of localization, we should provide as much as possible the content of the program that is closely linked to the current statement. Therefore, an easy way is to provide the control flow chart of function which the current reviewed statements belong, and then add heuristic information on the CFG. But this approach faces a dilemma: the size of the current statement's function is likely to be too large, leading to a large amount of information given in the visualized graphics model and not easy for the programmer to identify whether the current statement is correct. In addition, the size of the current function is likely to be too small (for example, only a few lines of code), which in turn leads to too little information available, and it is not conducive to the fact that the programmer to identify the current statement.

Therefore, we proposed three strategies for visualization in accordance with the size of the function which the current reviewing statement contained:

1) When the function scale is moderate(for example, the number of basic blocks is between a upper bound $\top$, and a lower bound $\bot$), the control flow chart of the function is given directly, and the dynamic behavior information (such as the value of the variable) is added to the corresponding program execution path in the control flow chart.
2) When the size of the function is too large (for example, the number of basic blocks above a certain upper bound to $\top$), the dynamic behavior (such as variable values, etc.) are added to the control flow chart corresponding to the program execution path to be displayed.
3) When the size of the function is too small (for example, the number of basic blocks below a lower bound to $\bot$), directly to present the function of the control flow chart. Meanwhile, for the function of the upper level call function analysis, analysis strategy which is now given the three strategies. Finally, the dynamic behavior information (such as variable values, etc.) added to the control flow chart corresponding to the program execution path to be displayed.

The choice of the above strategies 1 to 3 is based on the scale of the function, where the upper bounds ($\top$) and the lower bounds ($\bot$) are two empirical values. Then we elaborate the implementations of suspicious statement auxiliary debugging information visualization algorithm with three strategies.

The following Algorithm 1 provides a specific implementation process.

In Algorithm 1, the input is the source code P, current the statement $ExaStmt$ under reviewed, a set of dynamic information $TraceInfo$ tracked during program running, whose data structure contains a queue and the variable storage list as shown in Figure 1 and a set of system settings parameters

Algorithm 1 Visualizing auxiliary information for debugging

Input:
$< P, ExaStmt >$;
$< TraceInfo >$;
$< \top, \bot, \mathcal{W} >$.
Output:
$DOTFile$.

```
1:  get fun where ExaStmt ∈ fun;
2:  size ← sizeof(fun);
3:  if size ∈ [⊥, ⊤] then        /* strategy 1 */
4:      Aux ← S1_Gen(fun, ExaStmt, TraceInfo, W);
5:  end if
6:  if size ∈ (0, ⊥) then        /* strategy 2 */
7:      f ← fun ;
8:      FunList ← ∅ ;
9:      while sizeof(f) ∈ (0, ⊥) ∧ FunList.size < W do
10:         FunList.add(f);
11:         f ← getCallerof(f);
12:     end while
13:     Aux ← S2_Gen(FunList, ExaStmt, TraceInfo, W);
14: end if
15: if size ∈ (⊤, ∞) then        /* strategy 3 */
16:     Aux ← S3_Gen(fun, ExaStmt, TraceInfo, W);
17: end if
18: DOTFile ← GenerateDOT(Aux);
19: return DOTFile;
```

$< \top, \bot, \mathcal{W} >$ which represent the maximum number of nodes in the upper bound, lower bound, and final display window. First we select the function fun which the current examined statement $ExaStmt$ belongs, measure the size (basic block number) of the function (fun). Then comparing the size of (fun) with the thresholds $(\bot, \top)$ set in advance, we choose different strategies accordingly, such as strategy 1 (lines 3 to 5), strategy 2 (lines 6 to 14) or strategy 3 (lines 15 to 17). We use a single function to denote the detailed process of the above strategies, respectively. Finally, the final result has been transformed to **DOT** format which can be visualized for guiding programmer's debugging activities.

The functions $S1\_Gen$, $S2\_Gen$ and $S3\_Gen$ in Algorithm 1 extract the basic blocks and predicates associated with the current reviewed statements, conduct the data dependencies analysis of variables, and add the dynamic information obtained in the execution of the program to the relevant basic blocks and predicates of the nodes of the control flow graph of the program. As we all known that slicing can be useful to improve developer productivity[17], [18], especially for developers dealing with very complex or unfamiliar code. So we applied backward- and forward- slice techniques in functions $S1\_Gen$, $S2\_Gen$ and $S3\_Gen$, respectively. In addition, in order to avoid the final display of too much auxiliary information, we calculate the distance between the node and the current execution path, remove the nodes on the original control flow diagram that are far from the current

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

execution path. The distance can be calculated equation 1 as follow.

$$dis(node^*, path) = \min\|path^*\| \quad \text{where} \\ \forall\, path' = <node^*, \cdots, node> \wedge node \in path \tag{1}$$

Where $path^*$ represents a node from $node^*$ node to a series of nodes to reach the last node $node$, and there is only one node $node$ in $path^*$ belongs to the current execution path $path$. So the above distance can be regarded as the shortest path.

Next, we will describe the three functions $S1\_Gen()$, $S2\_Gen()$, and $S3\_Gen()$ in turn with Algorithm 2, Algorithm 3 and Algorithm 4, respectively.

Algorithm 2 first constructs a segmental control flow chart ($SegCFG$) from the function entry to the current examined statement ($ExaStmt$) at line 1. Then, a backward slice, the maximum size of which is limits by the parameter $\mathcal{W}$, was computed with respect to $ExaStmt$ (line 2). The reference variable is then calculated for each dependency statement in the backward slice, then the values of the corresponding variables, extracted from the trace information $TraceInfo$, are added to the variable position of the corresponding node in the control flow segment $SegCFG$ (line 3 to 11). In the algorithm 2, the variable $node.stmt.v.values$ means that there is a statement $stmt$ which has a reference variable v. Similarly, the function $TraceInfo.getValue(node, v)$ means that the object $TraceInfo$ have a method $getValues$, which is to get the value of variable (v) in the corresponding node $(node)$.

Algorithm 2 $S1\_Gen(fun, ExaStmt, TraceInfo, \mathcal{W})$

```
1:  SegCFG ← ConstructSegCFG(fun, ExaStmt);
2:  BWS ← BackwardSlice(SegCFG, ExaStmt, W);
3:  for each stmt ∈ BWS do
4:      VAR ← Ref(stmt);
5:      for each node ∈ SegCFG do
6:          for each v ∈ VAR ∧ v is referenced by node do
7:              node.stmt.v.values ←
8:                      TraceInfo.getValues(node, v);
9:          end for
10:     end for
11: end for
12: return SegCFG;
```

Next, the function $S2\_Gen()$ is described in algorithm 3. First, the control flow graphs of all the functions in the function list are calculated, and the control flow pieces are concatenated into a large control flow picture segment in the order of invocation and invocation (line 1 to 6). The method $fun.getCaller()$ returns the statements which current function calls. The subsequent work (line 7 to 16) is similar to that of the function $S1\_Gen()$.

Algorithm 4 describes the strategy of extracting and presenting the auxiliary debugging information when the size of the current function of the review statement is very large and exceeds the limit threshold.

Algorithm 3 $S2\_Gen(FunList, ExaStmt, TraceInfo, \mathcal{W})$

```
1:  SegCFG ← ConstructSegCFG(FunList[0], ExaStmt);
2:  for i from 1 to sizeof(FunList) do
3:      CallerStmt ← FunList[i − 1].getCaller();
4:    SegCFG ←ConstructSegCFG(FunList[i], CallerStmt);
5:      SegCFG ← SegCFG.Connect(SegCFG′)
6:  end for
7:  BWS ← BackwardSlice(SegCFG, ExaStmt, W);
8:  for each stmt ∈ BWS do
9:      VAR ← Ref(stmt);
10:     for each node ∈ SegCFG do
11:         for each v ∈ VAR ∧ v is referenced by node do
12:             node.stmt.v.values ←
13:                     TraceInfo.getValues(node, v);
14:         end for
15:     end for
16: end for
17: return SegCFG;
```

In Algorithm 4, the function $fun()$ segmental control flow diagram $SegCFG$ is first calculated (line 1). Then with the tracking information of program run obtained, the path fragment $SegPath$, ended with current review statement $ExaStmt$, is constructed whose length does not exceed $\mathcal{W}$ (line 2). As the first step in the control flow diagram fragment is too large. If the direct display to the programmer these complex information, then it will not be conducive to the programmer quickly get accurate, refined information. Therefore, we need to delete the initial control flow diagram fragment, remove the node with little relationship with current

Algorithm 4 $S3\_Gen(fun, ExaStmt, TraceInfo, \mathcal{W})$

```
1:  SegCFG ← ConstructSegCFG(fun, ExaStmt);
2:  SegPath ←TraceInfo.ConstructPath(SegCFG, ExaStmt, W);
3:  for each node ∈ SegCFG do
4:      if dis(node, SegPath) > |Min_Distance| then
5:          SegCFG.remove(node);
6:          SegCFG.delete(aEdges) where
7:              < node, n′ > ∨ < n′, node >∈ aEdges ;
8:      end if
9:  end for
10: BWS ← BackwardSlice(SegCFG, ExaStmt, W);
11: for each stmt ∈ BWS do
12:     VAR ← Ref(stmt);
13:     for each node ∈ SegCFG do
14:         for each v ∈ VAR ∧ v is referenced by node do
15:             node.stmt.v.values ←
16:                     TraceInfo.getValues(node, v);
17:         end for
18:     end for
19: end for
20: return SegCFG;
```


Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

failed execution. For each node in the control flow diagram $SegCFG$, we calculate the distance from the execution path fragment $SegPath$ to the node. If the distance exceeds the threshold set by the system $Min\_Distance$, remove the node from $SegCFG$ and delete the edge associated with the node (line 3 to 9). The subsequent work (line 10 to 19) is similar to that of the function $S1\_Gen()$ and $S2\_Gen()$. That is, with respect to the current review statement $ExaStmt$, the backward slices with the size of $\mathcal{W}$ were calculated based on the pruned control flow chart, and then extract the actual variables in the implementation of variable values, add the variable value information to the node for subsequent visualization.

After answering the research questions (**RQ1** and **RQ2**), that is, to identify the debugging auxiliary information and the debugging auxiliary information display two questions. Next, we will discuss the third research question (**RQ3**) which focuses on making full use of the programmer's early work during debugging.

III. Suspicious Statements Recommendation by Debugging Feedback

In this section, we will discuss how to dynamically adjust the suspiciousness of statements based on the feedback by programmer to answer the last research question (**RQ3**) followed by some basic assumptions.

A. Basic Assumptions

In the existing fault locating scenario applying SBFL technologies, the programmers determine whether the current examined statement in a fault locating report (often a rank list of statements with suspiciousness descending order) is an erroneous statement or not with their personal experience when debugging. Normally, they will repeat the process until all statements are examined. However, the programmer's judgment on the previously examined statements has not been fully utilized in the subsequent work during the debugging process. We believe that the judgment of the programmer can guide the adjustment of the order in which subsequent statements to be examined. To this end, a strategy of dynamic adjustment statement suspiciousness based on debugging feedback is proposed. The strategy is based on the following three assumptions:

Assumption 1: Each judgment for the reviewed statements made by the programmers is always right.

Assumption 2: If the programmer determines that the current reviewed statement (s) is correct, the suspiciousness of the statements in the backwards-slice of the s statement is overestimated and should be lowered suspicious of these statements.

Assumption 3: If the programmer determines that the current reviewed statement (s) is faulty, the suspiciousness of the statements in the forward-slice of the s statement is overestimated and should be lowered suspicious of these statements.

The assumption 1 is also called **skilled programmer** hypothesis. Because the judgments of the examined statements by the programmers at present are mainly based on the subjective of the experience, the complexity of human mental work determines that the judgment result may be wrong and the error probability varies from person to person. Although we can build a dynamic feedback model based on the probability of debugging personnel, but that will significantly increase the complexity of the debugging model. In general, the probability of an experienced programmer error is low. For the sake of analysis, we assume that developers are skilled, and their judgment during debugging will not go wrong. In other words, Assumption 1 holds.

The existing statistical fault localization techniques usually presuppose that every statement in the program is likely to go wrong. It is well known from PIE (Propagation -Injection - Execution) model that software fault caused the program to enter an error internal state, and the error state may pass through the program execution path to the program exit[19]. During the propagation process, if a statement is wrong, then the suspicious value of the statement from the beginning of the program to the software fault, which associated with the software fault, is overvalued and should be lowed (Assumption 2); Correspondingly, if a statement is correct, then the suspiciousness of all the relevant statements from the statement to the program exit is also overestimated and should be adjusted to proper values (Assumption 3).

B. Dynamic recommendation for suspicious statement based on debugging feedback

Let us consider one statement identification scenario. We modified the suspiciousness of the remained statements with two strategies as follows:

(1) When the programmer identifies that the current reviewed statement s is correct, the suspiciousness of the remained statements (s) can be updated with the followed formula.

$$R_s^{'} = R_s \times (1 - \frac{R_s}{\sum_{s_i \in \Omega} R_i}) \tag{2}$$

where Ω is the backward slice of statement s, R_i is the suspiciousness of the statement s_i.

(2) When the programmer identifies that the current reviewed statement s is faulty, the suspiciousness of the remained statements (s) can be updated with the followed formula.

$$R_s^{'} = R_s \times (1 - \frac{R_s}{\sum_{s_i \in \Psi} R_i}) \tag{3}$$

where Ψ is the forward slice of statement s, R_i is the suspiciousness of the statement s_i.

Due to the huge scale and expensive computing of precise dynamic slices, we applied an improved dynamic slice algorithm, named limited preprocessing (LP) algorithm, which is proposed by Zhang et al. [20] for backward and forward slices computing. The LP algorithm is practical because it never runs out of memory and is also fast.

The algorithm 5 shows the steps of dynamic adjusting the suspiciousness. Statement S_{top}, the top element of the examining list, has the highest suspiciousness. The function

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 5 Feedback Based Suspiciousness Updating

Input:
$RankList$;
Output:
$newRankList$;
1: $< S_{top}, R_{top} > \leftarrow RankList.pop$;
2: $assert \leftarrow Feedback(S_{top})$;
3: **if** $assert$ is Right **then**
4: $\Omega \leftarrow BackwardSlice(S_{top})$;
5: $\rho \leftarrow \frac{R_{top}}{\sum_{s_i \in \Omega} R_i}$;
6: **for** $each\ s_i \in \Omega \wedge s_i \in RankList$ **do**
7: $R_i \leftarrow R_i \times (1 - \rho)$;
8: $RankList.Update(s_i, R_i)$;
9: **end for**
10: **end if**
11: **if** $assert$ is Wrong **then**
12: $\Psi \leftarrow ForwardSlice(S_{top})$;
13: $\rho' \leftarrow \frac{R_{top}}{\sum_{s_i \in \Psi} R_i}$;
14: **for** $each\ s_i \in \Psi \wedge s_i \in RankList$ **do**
15: $R_i \leftarrow R_i \times (1 - \rho')$;
16: $RankList.Update(s_i, R_i)$;
17: **end for**
18: **end if**
19: $newRankList \leftarrow RankList.Sort()$;
20: **return** $newRankList$;

Algorithm 6 Group Feedback Based Suspiciousness Updating

Input:
$RankList$;
Output:
$newRankList$;
1: $< S_{top}, R_{top} > \leftarrow RankList.pop$;
2: $< Good, Bad > \leftarrow Feedback(S_{top})$;
3: **for** $each\ S_{good} \in Good \wedge S_{good} \in RankList$ **do**
4: $\Omega \leftarrow BackwardSlice(S_{good})$;
5: $\rho \leftarrow \frac{R_{good}}{\sum_{s_i \in \Omega} R_i}$;
6: **for** $each\ s_i \in \Omega \wedge s_i \in RankList$ **do**
7: $R_i \leftarrow R_i \times (1 - \rho)$;
8: $RankList.Update(s_i, R_i)$;
9: **end for**
10: **end for**
11: **for** $each\ S_{bad} \in Bad \wedge S_{bad} \in RankList$ **do**
12: $\Psi \leftarrow ForwardSlice(S_{bad})$;
13: $\rho' \leftarrow \frac{R_{bad}}{\sum_{s_i \in \Psi} R_i}$;
14: **for** $each\ s_i \in \Psi \wedge s_i \in RankList$ **do**
15: $R_i \leftarrow R_i \times (1 - \rho')$;
16: $RankList.Update(s_i, R_i)$;
17: **end for**
18: **end for**
19: $RankList \leftarrow RankList.Remove(Good, Bad)$;
20: $newRankList \leftarrow RankList.Sort()$;
21: **return** $newRankList$;

$Feedback(S_{top})$ returns the judgment on the current statement S_{top} made by the programmer (line 2). When the programmer determines that the current statement S_{top} is correct, the suspiciousness of all the statements in the S_{top}'s backward slice should be lowered(line 3 to 10). That is, we first calculate the S_{top}'s backward slice Ω, then calculate the ratio (ρ) of suspiciousness of S_{top} with the sum of all the statement's suspiciousness in backward slice, and the new suspiciousness of remained statements are finally updated with formula 2. Similarly, When the programmer determines that the current statement S_{top} is wrong, the suspiciousness of all the statements in the S_{top}'s forward slice should be lowered(line 11 to 18).

C. An improved suspiciousness updating algorithm based on group feedback

However, the programmer can only confirm a statement at a time in the algorithm 5, which results in lower debugging efficiency. In the actual software debugging process, the programmer may confirm a set of statements at one time, which can be divided into two collections ($Good$ and Bad). The correct statements set was denoted as $Good$, and the faulty statements set was denoted as Bad. We need to further refine the algorithm 5 to deal with multiple statements feedback situations, as showed by the algorithm 6.

The role of line 3 to line 10, and line 11 to line 18 in the algorithm 6 are similar with the role of line 3 to line 10, and line 11 to line 18 in the algorithm 5. In algorithm 6, line 5 and line 13 obtain the ratio for adjusting suspiciousness of remained statements, respectively.

Unlike the algorithm 5, the algorithm 6 iterates over multiple statements which are identified by the programmer as a feedback with two sets $Good$ and Bad. In addition, the line 19 of the algorithm 6 is used to remove the identified statements in sets $Good$ and Bad from suspicious rank-list because there is no need to re-confirm.

Further analysis of the algorithm 5 and the algorithm 6, we have found that the order in which the two algorithms handle the feedback of the programmer may affect the sorted list of the final generated suspicious statements. At present, we do not even know clearly the extent of its specific impact. One of the main reasons is that the effect of these two algorithms depends on whether the judgment given by the programmer is correct, which makes it difficult for us to analyze theoretically and practically.

IV. Prototype Presentation

The intuition of behind our approach lies that the heuristic information of program running are helpful for understanding software logic and identifying the root cause of failure run. In addition, the programmer's previous judgments may be useful for the identification of successor faults. Lighten by the intuition, we first construct a visualized heuristic digram

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

based on the traced running of tests to help the programmer to identify the current examined entities(such as program statements), then adjust the previous fault-locating report for next fault identification progress based on the judgment made by the programmer previously. Figure 2 illustrates the overview of the proposed interactive debugging with heuristic information. Our framework mainly with two phases: Heuristic information visualization and Improving fault localization report dynamically.

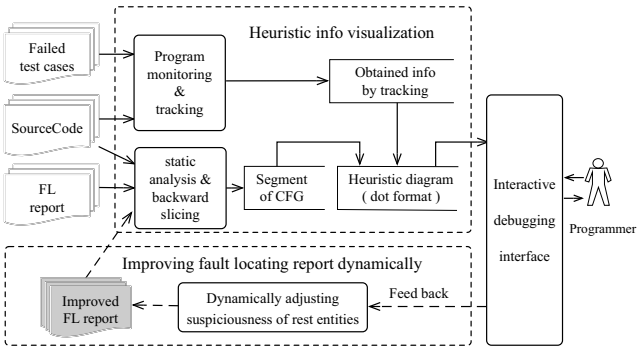

Fig. 2. Overview of the interactive debugging with heuristic information

Our approach is a cycle iterative processes mainly composed of three stages as follows:

1) **Information Collecting and Modeling**: This stage is mainly based on the program slicing and other analytical techniques to collect the auxiliary program information for locating faults, and then construct the model of the relationship between program entities.
2) **Model Analysis and Presentation**: This stage analyzes the relationship between the program entities in the relationship model, filters the program entities, and visualizes the heuristic information for fault locating.
3) **Dynamic Adjustment of Suspiciousness Based on Feedback**: According to the programmer's feedback, this stage dynamically adjusts the suspiciousness of the unexamined program entity to ensure that the most suspicious program entity is always recommended for review by the programmer first during the software debugging process.

As Figure 2 shows, the input to our approach is the program source code with a set of failed test cases and the initial fault localization report. First, running information was tracked and modeled when the program reviewed runs with failing test cases. Second, program slices will be computed with the highest suspicious program entity as criteria, which will be under reviewed by the programmer. Then, a segment of the control flow graph (segment-CFG) will be built with static program analysis considering the slices. Next, heuristic information formed as dot formates can be computed combining with both segment-CFG and the tracked dynamic information. Then, the programmer can make a judgment that whether the current reviewed entity is faulty or not. Meanwhile, the judgment will be fed back to our framework for the subsequent fault locating. In detail, our framework will adjust the suspiciousness of the remained entities and generate a new fault localization report, which is useful for the identified of faults.

V. Related work

Our work was inspired by the PIE model which illustrated the defect propagation path[19]. Generally, statistical fault localization techniques construct a rank list composed of program entities which guide the developer to identify the elements to be faulty or not [1], [2], [21]. Researchers are just trying to provide developers with an intuitive bug report[2]. Unfortunately, few of them can provide heuristic information to help the developer to identify the root cause of the faults.

It is a known fact that program slices present the dependencies between program entities [17]. It is considered to be helpful for program understanding and fault locating [4]. In the early years, Zhang et al. proposed three dynamic slice algorithms which are more precise [20]. In this paper, we selected one of these algorithms called LP as the basis for our forward slicing and backward slicing.

Interactive debugging is the closest means to real software maintenance practices. Researcher fight countless ways for improving the efficiency and effectiveness of debugging [3], [6], [9]. Early, Myers et al. propose a method that displaying the related data structure while debugging [6]. Hao et al. applied a check-point mechanism to SBFL approach which improved the effectiveness compared with existing fault localization approaches [3]. Recently, Lin et al. proposed a lightweight human feedbacks based debugging approach which can recommend suspicious execution trace [9].

Unlike the above, our approach traces one failure execution of the program, then provide not only the data dependency information, but also the control dependency information, and provide a heuristic graph to help the developer identifying bugs. We also provided a feedback based debugging approach which can adjust the suspiciousness of program entities adaptively.

VI. Conclusions and future work

The motivation of this study was to provide heuristic information that can help the developer to identify bugs in real program debugging more efficiently. To this end, we first proposed a practical method of modeling and visualizing heuristic information extracted from runtime traces. Furthermore, in order to avoid showing too much or too little inspiration information to the developers, we put forward three strategies to help ensure that the information is reasonable. Then, considering the full use of developer feedback, we proposed an effective algorithm to dynamically improve and reduce the suspicious degree of the corresponding program statement. In addition, we proposed a kind of reinforcement algorithm to adjust the statements' suspicious in batches according to a set of feedback from the developer, thus increasing the efficiency of program debugging.

Our work is preliminary but it is very practical in program debugging. We need further discussion of the advantages and

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.

weakness of our approach. For example, our approach requires repeated computation of slices, including forward slices and back slices. This will cost a lot of computation time. Thus, increasing the waiting time for the actual debugging. In the future, there will be some aspects to improve and study. (1) our algorithms should be improved to reduce the slice calculation time. For example, some dependencies can be calculated and stored in the central information repository in advance, which is obtained directly when the algorithm calculates slices later. (2) besides algorithm improving, we also need to find a way to get the optimal width of code examination window. The reason lies that human attention is very limited. During program debugging, too much or too little information is not conducive to the developer to make the right decision. (3) a meaningful work is to mine the history of software development for obtaining the defect related clues. As we all know, the are many source control systems such as github, which contains the information throughout software development. We can make use of the information stored in the software repository, such as commitment, bug fixing, etc., to help build the inspired information about the current debugging version.

ACKNOWLEDGMENT

This work was supported in part by awards from the National Natural Science Foundation of China under Grant Nos. 61502497, 61602267 and 61673384, the Guangxi Key Laboratory of Trusted Software Research Plan under Grant Nos. KX201530 and KX201532, and the National Natural Science Foundation of Guanxi under Grant No 2015GXNSFDA139038.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 1–41, 2016.

[2] J. A. Jones, "Fault localization using visualization of test information," in *Proceedings of the 26th International Conference on Software Engineering(ICSE 2004)*. Edinburgh, Scotland, UK: IEEE Computer Society, 2004, pp. 54–56.

[3] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive fault localization using test information," *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, 2009.

[4] V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta, "A system for debugging via online tracing and dynamic slicing," *Software: Practice and Experience*, vol. 42, no. 8, pp. 995–1014, 2012.

[5] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, vol. 25, no. 1, pp. 83–110, 2017. [Online]. Available: http://dx.doi.org/10.1007/s11219-015-9294-2

[6] B. Myers, "Displaying data structures for interactive debugging," Master's thesis, Massachusetts Institute of Technology, Cambridge, 1980.

[7] X. M. Wang, S. C. Cheung, W. K. Chan, and Z. Y. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. Los Alamitos, CA: IEEE computer society, 2009, pp. 45–55.

[8] T.-D. B. Le, D. Lo, and F. Thung, "Should i follow this fault localization tools output?" *Empirical Software Engineering*, vol. 20, no. 5, pp. 1237–1274, 2015.

[9] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[10] S. Ali, J. H. Andrews, T. Dhandapani, and W. T. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*. Los Alamitos, CA: IEEE Computer Society, 2009, pp. 76–87.

[11] M. A. Alipour, "Automated fault localization techniques: a survey," Technical report, Oregon State University, Corvallis, Oregon, Tech. Rep., 2012.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Empirical effectiveness evaluation of spectra-based fault localization on automated program repair," in *Proceedings of the 37th Annual International Computer Software and Applications Conference (COMPSAC 2013)*. Los Alamitos, CA: IEEE, 2013, pp. 33–42.

[13] W. E. Wong and V. Debroy, "Software fault localization," Tech. Rep., 2009.

[14] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010)*. New York, USA: ACM, 2010, pp. 73–84.

[15] T. Y. Chen, P. Hu, H. Li, and T. Tse, "An enhanced flow analysis technique for detecting unreachability faults in concurrent systems," *Information Sciences*, vol. 194, pp. 254–269, 2012.

[16] B. Jiang, K. Zhai, W. K. Chan, T. H. Tse, and Z. Y. Zhang, "On the adoption of mc/dc and control-flow adequacy for a tight integration of program testing and statistical fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 897–917, 2013.

[17] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[18] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI 1990)*, vol. 25. New York, USA: ACM, 1990, pp. 246–256.

[19] J. M. Voas, "Pie: a dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, 1992.

[20] X. Y. Zhang, M. X. Lin, and K. Yu, "An integrated bug processing framework," in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. Los Alamitos, CA: IEEE computer society, 2012, pp. 1469–1470.

[21] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, March 2014.

Authorized licensed use limited to: Nan Tong University. Downloaded on January 21,2025 at 13:19:21 UTC from IEEE Xplore. Restrictions apply.