From Predicate Testing to Identify Fault Location for Safety-Critical Software

Yong Wang\textsuperscript{a,b}, Qiansong Wang\textsuperscript{a,*}, Guifu Lu\textsuperscript{a}, Zhiqiu Huang\textsuperscript{c}, Bingwu Fang\textsuperscript{c}, Yong Li\textsuperscript{c}, and Weiwei Li\textsuperscript{c}

\textsuperscript{a}School of Computer and Information, Anhui Polytechnic University, Wuhu, 241000, China
\textsuperscript{b}Key Laboratory of Safety-Critical Software, Nanjing University of Aeronautics and Astronautics, Nanjing, 210000, China
\textsuperscript{c}College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China

Abstract

Statistical fault localization is one of the essential tasks of program debugging, and it has shown that the evaluation history of predicates may disclose important clues about the root cause of failures. However, especially for safety-critical software, there exists evaluation bias using same granularity to measure simple predicates and complex compound predicates. Intuitively, we should use fine-grain predicates to evaluate the suspiciousness of complex compound predicates and reduce the evaluation bias. In this paper, we propose a novel predicate fault localization technique from predicate testing to identify fault location. Based on the predicate fault model, we first generate constraint sets for each predicate and then calculate the suspiciousness of predicates by evaluating their constraint sets. Finally, we sort the suspicious predicates by their suspiciousness. Our preliminary results show that our approach can significantly improve fault predicate absolute ranking.

Keywords: fault localization; predicate testing; debugging

(Submitted on May 23, 2018; Revised on July 11, 2018; Accepted on August 11, 2018)

© 2018 Totem Publisher, Inc. All rights reserved.

1. Introduction

Safety-critical software as a component of safety-critical systems may configure several parameters and interact with other components. Therefore, safety-critical software usually contains complex logics and are more complicated than other software [1]. Ensuring such software do not fail is usually accomplished by testing, which can reveal software failures by executing the program over a set of test cases. However, only detecting program failures via testing is inadequate. The programmers must continue to localize the fault(s) and fix them by utilizing information collected during testing.

A real aircraft collision avoidance system (tcas) is a typical safety-critical software that takes 12 input parameters and produces an output that can be either 0, 1, or 2. Figure 1 shows a logic bug in tcas. Variables enabled should set the value of logic expression: \textit{High \_Confidence \&\&(Own \_Tracker \_Alt \_Rate <= OLEV) || (Cur \_Vertical \_Sep > MAXALTDIFF)}.

However, we accidentally write the second logic operation '&&' as '||':

Automated fault localization has been studied for decades for functional bugs. Among these techniques, statistical fault localization is one of the most effective [2]. Specifically, statistical fault localization collects program predicates, such as whether a predicate is executed, during both failed tests and successful tests, and then it uses statistical models to generate a predicate ranking list that orders suspicious predicates by their suspiciousness [3-4].

Unlike other systems, there are some predicates contained in complex logic, which are more likely to be bugs due to their complex logic, and others are not in safety critical software [1]. Therefore, this would lead to evaluation bias for evaluating suspiciousness of these predicates via statistical fault localization techniques. A feasible solution is to convert a

* Corresponding author.
E-mail address: YongWang@ahpu.edu.cn
complex predicate into a series of simple predicates and measure the simple predicates to obtain the suspiciousness of complex predicates. Hence, the problem is properly decomposing a predicate to better measure the suspiciousness for complex predicates. Inspired by predicate fault models, we decompose a complex predicate from four different predicate faults models’ views. Intuitively, if a predicate contains root fault(s), it cannot have low suspiciousness for all different measurement views. Therefore, we can take the largest suspiciousness among them as the suspiciousness of the predicate.

The paper is structured as follows. Section 2 describes some background for our approach. Section 3 presents our approach, followed by an empirical evaluation in Section 4 and related works in Section 5. Section 6 concludes the paper and provides some direction for future works.

2. Background

2.1. Predicate Testing

In this section, we provide a detailed example to illustrate predicate testing. The program segment in Figure 1 is excerpted from the program tcas v3 [5]. The tcas is a safety-critical system, which has about 141 lines of C code (LOC) and is an aircraft collision avoidance system designed to reduce the incidence of mid-air collisions between aircraft. The system contains many predicates that encode the logical of the program. We give some definitions about predicate testing as follows:

```c
int alt_sep_test()
{
    bool enabled, tcas_equipped, intent_not_known;
    bool need_upward_RA, need_downward_RA;
    int alt_sep;
    enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) ||
    (Cur_Vertical_Sep > MAXALTDIFF);
    /* enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
       && (Cur_Vertical_Sep > MAXALTDIFF); */
    tcas_equipped = Other_Capability == TCAS_TA;
    intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
    ...
}
```

**Figure 1. A bug in tcas**

**Definition 1** Predicate. A predicate is an expression that evaluates the Boolean value.

Predicates contain a Boolean variable, a non-Boolean variable that contains >, <, =., <., >., !., and Boolean function calls. The internal structure of a predicate is connected by logical operators, and the operators are as follows:

- ∨, the or operator.
- ¬, the negation operator.
- ∧, the and operator.

**Definition 2** Clause. A clause is a single predicate with no logical operators.

In the example,

\[ enabled = \text{High\_Confidence} \lor (\text{Own\_Tracked\_Alt\_Rate} \leq \text{OLEV}) \lor (\text{Cur\_Vertical\_Sep} > \text{MAXALEDIFF}) \]

is a predicate, which contains three clauses:

\[
\begin{align*}
A &= \text{High\_Confidence} \\
B &= \text{Own\_Tracked\_Alt\_Rate} \leq \text{OLEV} \\
C &= \text{Cur\_Vertical\_Sep} > \text{MAXALEDIFF}
\end{align*}
\]
In Figure 1, the predicates are intentionally written as follows by SIEMENS researchers to simulate a type of logic operator fault model.

\[
\text{enabled} = \text{High} \_ \text{Confidence} \land (\text{Own} \_ \text{Tracked} \_ \text{Alt} \_ \text{Rate} \leq \text{OLEV} \lor (\text{Cur} \_ \text{Vertical} \_ \text{Sep} > \text{MAXALEDIFF}))
\]

Lau et al. proposed a fault model hierarchy, as shown in Figure 2. Each three-letter acronym represents a fault model that can occur in a predicate; for example, \( \text{LIF} \) is the literal insertion fault (\( \text{LIF} \)) where a clause is added into the predicate. The detailed information can be found in [6].

![Figure 2. Lau’s fault hierarchy](image)

Generally, there mainly exist four predicate fault models as follows, whose details can be found in [7].

- Boolean operator fault model (\( \text{BOFM} \)).
- Relational operator fault model (\( \text{ROFM} \)).
- Arithmetic expression fault model (\( \text{AEFM} \)).
- Missing Boolean fault model (\( \text{MBFL} \)).
- Extra Boolean variable fault model (\( \text{EBVFL} \)).

To test these fault models, there exist many predicate testing criteria to generate test cases. For a predicate \( p_c \) and its fault version \( p_f \), predicate testing mainly focuses on generating a test suite \( T \) such that there exists at least one test case \( t_i \) to evaluate a different output value. There exist various testing criteria, such as \( \text{BOR} \), \( \text{BRO} \), and \( \text{BRE} \) [7].

2.2. Statistical Fault Localization

Statistical fault localization (\( \text{SFL} \)) techniques, which exploit program spectra and calculate the likelihood of each program component for a buggy program being a fault, is a lightweight approach [2]. The key for an \( \text{SFL} \) is the suspiciousness measurement, which is used to calculate the suspiciousness score of program components. Several similarity measurements may be used to compute suspiciousness. Given program spectra \( A \) and Result Vector \( e \), which represents the results of each execution, the coverage information can be calculated as shown below:

- \( n_{00}(s) \) is the number of not covered components \( s \) in successful executions.
- \( n_{01}(s) \) is the number of not covered components \( s \) in failed executions.
- \( n_{10}(s) \) is the number of covered components \( s \) in successful executions.
- \( n_{11}(s) \) is the number of covered components \( s \) in failed executions.
- \( n_{11}(s) \) is sum of \( n_{11}(s) \) and \( n_{10}(s) \).
- \( n_{10}(s) \) is sum of \( n_{10}(s) \) and \( n_{00}(s) \).
- \( n_{11}(s) \) is sum of \( n_{00}(s) \) and \( n_{11}(s) \).

Four similarity measurements of well-known fault localization techniques, Jaccard, Tarantula, Ochiai, and Dstar, are defined as below:
\[ \text{Jaccard}(s) = \frac{n_1(s)}{n_0(s) + n_1(s)} \]  

(1)

\[ \text{Tarantula}(s) = \frac{(n_u(s) \times n_i(s))}{n_i(s) \times n_u(s) + n_i(s) \times n_j(s)} \]  

(2)

\[ \text{Ochiai}(s) = \frac{n_1(s)}{\sqrt{(n_i(s) \times n_u(s))}} \]  

(3)

\[ \text{Dstar}(s) = \frac{(n_i(s))^2}{n_i(s) + n_0(s)} \]  

(4)

3. Our Approach

3.1. Framework

Our approach focuses on localizing the predicate fault based on statistical techniques. The framework is shown below and consists of three main steps: (1) Instrument and program spectrum collection: In this step, we first generate constraint sets for each predicate based on predicate fault modes, and then we instruct different instrument codes for each predicate in the buggy program. Finally, we collect predicate program spectra (line 1-8). (2) Suspiciousness measurement: In this step, we first measure suspiciousness for elements of constraint sets, which are broken down for each predicate. Then, we generate suspiciousness for each predicate by obtaining the maximum of the suspiciousness of elements of constraint sets belonging to the predicate (line 9-17). (3) Fault ranking list generation: In this step, we sort the predicates by their suspiciousness (line 18-19). In the three steps, we focus on describing the detail of two key steps in the following subsections.

**Algorithm 1** Our approach framework

**Input**: Software under testing SUT, suspiciousness metric \( \rho_b \) and test suite \( \Gamma \)

**Output**: Predicate ranking list \( \Omega \)

1: //Step 1: Instrument and program spectrum collection
2: \( P_s \leftarrow \text{genPredicateSet}(SUT) \)
3: for each \( p' \) in \( P_s \) do
4: \( \langle p_{\text{source}}, p' \rangle \leftarrow \text{generateConstraint}(p) \)
5: \( P_{\text{max}} = \text{max} \cup \langle p_{\text{source}}, p' \rangle \)
6: end for
7: \( SUT \leftarrow \text{Instrument}(SUT, P_{\text{max}}) \)
8: \( \langle \langle A, P_s, e \rangle \leftarrow \text{run_program}(SUT) \rangle \)
9: //Step 2: Suspiciousness measurement
10: for \( j = 0 \) to \( |\Delta| \) do
11: \( \text{setValues}(n_{\text{bo}}(j), n_{\text{bi}}(j), n_{\text{bd}}(j), n_{\text{bu}}(j)) \)
12: \( S[j] \leftarrow \text{Dstar}(n_{\text{bo}}(j), n_{\text{bi}}(j), n_{\text{bd}}(j), n_{\text{bu}}(j)) \)
13: end for
14: \( S_{\phi} \leftarrow \phi \)
15: for each \( p' \) in \( P_s \) do
16: \( S_j[p'] \leftarrow \text{Max}(S[j], A, p') \)
17: end for
18: //Step 3: Fault ranking list generation
19: \( \Omega \leftarrow \text{sort}(S_{\phi}) \)
20: return \( \Omega \)

3.2. Constraint Set Generation

A predicate constraint \( C \) for a predicate \( p_r \), which contains \( n \) clauses, is a sequence of \((n+1)\) BR symbols, one for relational expression or each Boolean variable in \( p_r \). Consider the predicate \( p_r: (r < s) \land b \lor (u \geq v) \), one of the constraints \( C_i \) can be \((=, t, >) \). A test case \( <r=1, b=true, s=1, u=1, v=0> \) can satisfy \( C_i \) for \( p_r \).
In order to simplify our ideas, we only consider the two predicate fault models (BOFM, ROFM). In the following, we give the four testing criteria, which can generate different constraint sets:

- **Clause testing criterion**: we decompose the predicate into multiple subclauses and measure the true and false constraints of each clause as a program entity.
- **Predicate testing criterion**: we take each predicate in the buggy program as a whole and measure its suspiciousness for the true constraint and false constraint.
- **BOR testing criterion**: we use BOR criterion to generate test constraints, which can guarantee the detection of single or multiple Boolean operator faults in the implementation of the complex compound predicate $p_i$.
- **BRO testing criterion**: we use BRO criterion to generate test constraints, which can guarantee the detection of a single or multiple relational operator or Boolean operator faults in the implementation of the complex compound predicate $p_i$.

Obviously, it is easy to generate a constraint set for clause testing criterion or Predicate testing criterion respectively. For instance, for a predicate $P_i = a \lor b$, its predicate constrain set is $\{p = t, p = f\}$, and its **Clause constrain set** is $\{a = t, a = f, b = t, b = f\}$.

Next, we illustrate the procedure of the generation BRO constraint set. The BOR constraint set is similar as the procedure of BRO constraint set generation. Let $A$ and $B$ be two sets, and the **Cross-Product** of the two sets $A$ and $B$ is defined as:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\}$$  \hspace{1cm} (5)

The **Onto-product** of the two sets $A$ and $B$ is defined as:

$$A \otimes B = \{(u, v) \mid u \in A, v \in B, \text{such that each element of } A \text{ appears at least once as } u \text{ and each element of } B \text{ appears once as } v.\}$$  \hspace{1cm} (6)

For example, let $A = \{t, =, >\}$ and $B = \{f, <\}$, then $A \times B = \{(t, f), (t, <), (\approx, f), (\approx, <), (>, f), (>\), (>\), <\), <\})$, and the one of $A \otimes B$ is $\{(t, f), (=, <), (>\), (>\), <\), <\})$. The BRO constraint set can be generated as Algorithm 2.

**Algorithm 2** BRO constraint set generation

**Input**: complex compound predicate $P_i$

**Output**: BRO constraint set $S_o$

1: //Step 1: Generate syntax tree of $P_i$
2: $AST(P_i) \leftarrow$ generateAST($P_i$)
3: //Step 2: Label each leaf node with the constraint set $S_n$. For each leaf node which is Boolean variable, $S_n = \{(t), (\approx), (\approx), (>\), (>\), <\}, S'_n = t, S''_n = f$: For each leaf node which is relational expression, $S_n = \{ (>\), (\approx), (\approx)\}$.
4: //Step 3: Compute the constraint set for the next higher node in the syntax tree $AST(P_i)$. If type of node $n$ is ‘and’ or ‘or’, let $n_1$, $n_2$ be its immediate successor; If type of node $n$ is ‘not’, let $n_1$ be its immediate successor. $S_n$, $S'_n$, represents constraint set for node $n_1$, $n_2$ respectively.
5: Computing constraint sets for each non-leaf node as following:
6: for each node $n$ do
7: if nodeType ($n$) $= \text{‘or’}$ then
8: $S_n' = S_n' \otimes S_n''$
9: $(S_n \times \{f_i\}) \cup (\{f_i\} \times S_n)$, where $f_i, f_i$ is any element in $S_n'$, $S_n''$, respectively.
10: end if
11: if nodeType ($n$) $= \text{‘and’}$ then
12: $S_n' = S_n' \otimes S_n''$
13: $(S_n' \times \{t_1\}) \cup (\{t_1\} \times S_n')$, where $t_1, t_2$ is any element in $S_n'$, $S_n''$, respectively.
14: end if
15: if nodeType ($n$) $= \text{‘not’}$ then
16: $S_n' = S_n''$
17: $S_n'' = S_n''$
18: end if
19: end for
20: return last $S_0$ in $AST(P_i)$;
3.3.3. Suspiciousness Measurement

For a predicate \( P_i \) and an execution \( E \), \( \text{cov}(P_i) \) is true iff \( P_i \) was observed to be true at least once during execution \( E \). Similarly, we define \( \text{cov}(c_i) \) as follows:

**Definition 3:** \( \text{cov}(c_i) \) is true if \( \text{cov}(c_i) \) satisfies its constraint at some point during the program execution.

For example, a constraint set for predicate \( P_i \) is \( S_i = \{ P_i = t, P_i = f \} \). \( \text{cov}(P_i = t) \) is true iff predicate \( P_i \) is executed and is evaluated as “true”. Similarly, \( \text{cov}(P_i = f) \) is true iff predicate \( P_i \) is executed and is evaluated as “false”. We can view a constraint for \( P_i \) as a single program component. Therefore, the suspiciousness score of the constraint can be measured using a suspiciousness metric. We use the Dstar technique to measure suspiciousness for a constraint. We modify the Dstar as follows:

**Definition 4:** \( \delta(c_i) \) is the suspiciousness score for constraint \( c_i \), and the suspiciousness of \( c_i \) be defined as:

\[
\delta(c_i) = \frac{(n_{11}(c_i))^2}{n_{01}(c_i) + n_{00}(c_i)} \quad (7)
\]

Where \( n_{11}(c_i) \) indicates \( \text{cov}(c_i) = \text{true} \) and the result of running for the buggy program is failed, \( n_{01} \) indicates \( \text{cov}(c_i) = \text{true} \) and the result of the program running is successful, and \( n_{01} \) indicates \( \text{cov}(c_i) = \text{false} \) or the running did not reach the predicate and the result of running for the buggy program is successful.

For a constraint set based on a certain test criterion, we can use the maximum of the suspiciousness score as the suspiciousness of predicate \( P_i \). The suspiciousness score based on a certain test criterion is defined as follows:

**Definition 5:** \( \delta(p_j) \). Letting constraint set \( C \) for \( P_i \) be \( C = \{ c_1, c_2, \ldots, c_m \} \), the \( \delta(p_j) \) can be defined as:

\[
\delta(p_j) = \max \{ \delta(c_1), \ldots, \delta(c_m) \} \quad (8)
\]

Borrowed from predicate fault models, we decompose a complex predicate from three different views. Intuitively, if a predicate contains a root bug, it should be suspicious from different measurement views. In other words, it cannot be low on all measures, so we can take the largest suspiciousness among them as the suspiciousness of the predicate. In our paper, we give four basic constraint sets, and we use the maximum score among those suspiciousness scores.

Let \( \delta_x(P_i), \delta_y(P_i), \delta_z(P_i), \delta_y(P_i) \) be the suspiciousness scores via constraint sets generated by the four testing criterions respectively. The final suspiciousness for predicate \( P_i \) is defined as follows:

\[
\delta(p_j) = \max \{ \delta_x(P_i), \delta_y(P_i), \delta_z(P_i), \delta_y(P_i) \} \quad (9)
\]

4. Experiments

4.1. Subject Programs

Our goal is to evaluate the effectiveness of our approach for safety-critical software. Therefore, we hope that subject programs are safety-critical systems. However, it is difficult to collect safety-critical software as subject programs. Hence, we select the Siemens programs as our subject program, which contains a safety-critical software tcas. We include 129 faulty versions of the Siemens test suite from SIR [5]. Table 1 describes the information of the subject programs and test suite that we use. In Table 1, column “LOC” represents executable code lines, “# of Versions” represents the number of faulty versions, “# of Pres” represents the number of predicates, and “% of Pre” represents the average percentage of predicates with respect to all statements.
4.2. Improvement Metric

Effectiveness metrics are an important way to perform accurate and objective comparisons. There are two main improvement metrics used to measure fault localization effectiveness in the fault localization field: PDG-based metric and Ranking-based metric. However, the above two standard effectiveness metrics for SBFLs normalize the fault rank with respect to the size of the program. Parnin et al. indicate that the better result evaluated by the two metrics did not help the programmer in the debugging process [8]. To measure the effectiveness of our approach, we define an improvement metric to compare an SFL with our approach, as shown in Formula 10 [9-10]:

$$\text{Im} \text{provement}_{a}^{B}(A, B) = \begin{cases} 0, & \text{if } B = 0 \text{ and } A = 0 \\ -100\% , & \text{if } B = 0 \text{ and } A > 0 \\ \frac{B - A}{B} \times 100\% & \end{cases}$$ (10)

4.3. Results

In our experiment, we first generate four constraint sets based on testing criteria. Then, we create instrumented program versions for those subject programs, including both the original and faulty versions. Program spectra is collected by running those instrument programs. Finally, we calculate the suspiciousness score for each predicate via their constraint sets and rank them. In the context of fault localization, we usually view the complex predicate as a whole to perform fault localization. For example, for predicate $P = a \& \& b || c$, we calculate the suspiciousness score for $p$ and $\hat{p}$ based on Dstar, which is called $P − Dstar$. Our approach is named PCS. If we use $Dstar$ as the base suspiciousness metric, our approach can be called $PCS − Dstar$. In order to evaluate the effectiveness of our approach, we use $P − Dstar$ as a baseline to compare our approach.

1) **RQ1: Overall effectiveness:** The goal of this research question is to determine whether our approach with basic sitting is effective. To do this, we run $PCS − Dstar$ and $P − Dstar$ on the 129 fault versions.

Letting $P − Dstar$ be the baseline approach, we use the effectiveness measurement as mentioned above to measure our approach improvement. Figure 3 shows the results of this research. From the total program versions, our approach shows the effectiveness over $P − Dstar$ for 59 program versions and demonstrates that it performs the same as $Dstar$ for 60 program versions and worse than $Dstar$ for 10 program versions. The results indicate that the average improvement is 17.2%. Therefore, our approach effectively improves the absolute ranking for complex logic problems and can be used in practice.

2) **RQ2: Individual effectiveness for each program:** To further verify whether the above results generally hold for all the programs, especially for safety-critical software $tcas$, we investigate the results of each individual program. The results are shown in Figure 4. Clearly, our approach performs better than $P − Dstar$ in the six programs. The average improvement for $tcas$ is better than those for the other four programs. Let us focus on the program $tcas$. The program $tcas$ is an aircraft collision avoidance system designed to reduce the incidence of mid-air collisions between aircraft. The program contained some complex fault predicates typical of a safety-critical system. By that factor, we can infer that our approach is suitable for other safety-critical software that contain complex compound predicates.

3) **RQ3: Whether our approach can be generalized to other suspiciousness metrics:** We also investigate whether our approach can be generalized to other suspiciousness metrics aside from $Dstar$. There exist various similarity metrics to measure the suspiciousness in fault localization. Recently, Pearson et al. demonstrated that all suspiciousness metrics are equally good through their experiments [11]. Therefore, we only use Ochiai, Jaccard, and Tarantula to answer this question.
Figure 3. Comparison of PCS-Dstar to P-Dstar

Figure 4. Comparison for different programs

We use the same faulty versions and perform the same experiment to evaluate the effectiveness for different suspiciousness metrics. The results are shown in Figure 5. As the results show, our approach based on different suspiciousness performed better than the SFL technique. The results illustrate two problems: (1) Our approach can be generalized to other suspiciousness metrics, and (2) although the effectiveness of different suspiciousness is different, there is no result that is superior to the others, which is a similar result as [11].

Figure 5. Comparison for different suspiciousness measurements
5. Related Works

There exist many studies that have been performed on software fault localization in the last two decades [2, 9-10, 12-13]. One of them measures the likelihood between program entities (such as statements, blocks, or predicates) with failures. The main idea is that certain coverage information for a suspiciousness program entity is correlated with program failures. Liblit et al. isolated faults in buggy programs with instrumented predicates at particular points [3]. They first calculate the probability that predicate \( P \) being True triggers failure (called \( Failure(P) \)) and the probability that running \( P \) triggers failure (called context\((p)\)). The predicates are ranked based on their Important score, which is modeled based on \( Failure(P) \) and \( Context(P) \). Liu et al. proposed a technique called SOBER to rank those suspicious predicates. They highlighted some predicates that can be executed as true more than once in a test for the buggy program [4]. Therefore, they compute 
\[
\pi(P) = n(t) / (n(t) + n(f)),
\]
which represents the probability that \( P \) is evaluated as True in each execution of a test case. The probability distribution of \( \pi(P) \) in failed testing is significantly different from that in passed testing.

In real programs, short-circuit may occur frequently in program running. This means that for a complex compound predicate with more than one single clause, if the first clause is enough to determine the result of the complex compound, the following clause will not be executed. A debugging through evaluation sequence approach was proposed by Zhang et al. [14]. Similarly, Arumuga et al. demonstrated that compound Boolean predicates are useful predictors for bugs and proposed a predicate fault localization using compound Boolean predicates [15]. In their experiments, qualitative and quantitative evidence show that statistical debugging techniques can be effectively applied to complex predicates, and that the resulting analysis provides improved results.

Predicate testing has already required to verify level a software as per the DO-178B standard for safety-critical systems [16-18]. Recently, Durelli et al. pointed out that “the vast majority of predicates in real programs have only one clause” [1]. They calculated the number of clauses in 400 811 predicates in 63 open-source Java programs. They found that 88.02% of the predicates had only one clause, 9.97% had two clauses, 1.29% had three clauses, 0.47% had four clauses, 0.11% had five clauses, and less than 0.15% had more than five clauses. However, the complex predicates are more prone to error due to their complexity. Therefore, measuring suspiciousness for simple predicates and complex predicates based on the same metric is not appropriate. Different from the above approach, we propose a novel fault predicate localization from predicate testing.

6. Conclusions

We have proposed a fault predicate localization technique based on predicate testing to identify fault predicate location for complex logic problems. We first generate predicate constraint sets from four different views. Then, we collect a constraint nine set-level program spectrum executed by the instrumented program based on execution by the test suites. Finally, we calculate suspiciousness scores for those predicates based on their suspiciousness of constraint sets and rank them by their suspiciousness scores. We evaluated the effectiveness of our approach by comparing it with the predicate=hit-Dstar technique. Preliminary experimental results revealed that our approach can perform better than the fault predicate for complex logic problems. In our future work, we will try to use more programs to validate the effectiveness of our approach, and we would also like to further combine our approach with other techniques, such as SOBER, to investigate their effectiveness.

Acknowledgements

This work was supported by the Anhui University Natural Science Fund Key Project (Grant No. KJ2018A0116, KJ2016A252, KJ2017A104), National Key Research and Development Program (Grant No. 2016YFB100), Project 61772270 supported by the NSFC of China, Anhui Natural Science Foundation (Grant No. 1608085MF147), Natural Science Foundation of Jiangsu Province (Grant No. BK20170809), and Safety-Critical Software Key Laboratory Research Program (Grant No. 1015-56XCA18164).

References