Priority Strategy of Software Fault Localization

Sun Jirong
School of Computer Science, Sichuan University, Chengdu 610065, China

Li Zhishu
School of Computer Science, Sichuan University, Chengdu 610065, China

Ni Jiancheng
School of Computer Science, Sichuan University, Chengdu 610065, China

Yin feng
College of Computer Science and Technology, Southwest University for Nationalities, Chengdu 610041, China

Abstract: To a given test case, fault localization has to be proceeded when its output is wrong. A novel method is presented to localize a fault. Firstly, by analyzing the relation between testing requirement and test cases that satisfying it, some assistant test cases are selected out. Then, program slice is introduced to reduce the searching domain based on priority, which has been evaluated according to the occurrences in the selected slices. Two procedures, refining and augmenting, are followed here to fault localization: in the refining phase, the most suspicious codes are checked step by step; in the augmenting phase, more codes will be gradually considered on the basis of direct data dependency. At last, experimental studies are performed to illustrate the effectiveness of the technique.

Keywords: test suite management; execution slice; dynamic slice; assistant slice; fault localization; direct data dependency;

1. Introduction

To improve the quality of a program, we have to remove as many defects as possible in it without introducing new bugs at the same time. However, locating a fault is a complex and time-consuming process.

A usual way is to use debugging tools. Break points must be set along a program execution and values of variables examined as well as internal states at each break point. This approach has two significant disadvantages: one is that it requires users to develop their own strategies to avoid examining too much useless information, and the other is that it cannot reduce the search domain by prioritizing code based on the likelihood of containing faults on a given execution path.

Program slicing is a program analyzing technique that reduces a program to those statements relevant to a particular computation. Mark Weiser introduced program slicing of error variable to exclude irrelevant statements thus to reduce the searching domain, but the slice was still too large [1]. Furthermore, Li Bixin proposed firstly to construct the forward slice of the input and backward slice of the output, and then to obtain the intersection of these two slices, but too many codes left were to be examined [2]. An execution slice-based technique as reported in [3] can be effective in locating some program bugs, but not on others especially those in the code that is executed by both the failed and the successful tests. Another problem is that even if a bug is in the dice obtained by subtracting the execution slice of a successful test from that of a failed test, there may be still too many codes that need to be examined.

Much information is available to help us localize a fault after software testing, such as testing requirements and their associated test set, test result analysis etc. While most fault location techniques have not taken these into consideration.

In this paper, we present a novel method to locate a bug based on code priority strategy and program slicing technique. The next section details how to utilize the relation between test cases and the corresponding requirements to obtain the assistant information in fault localization. In section 3 we use a sample program to illustrate how to manage the test suite. Section 4 demonstrates our code prioritization methodology. In section 5, we present the key algorithms, refining and augmenting, to locate a fault based on code priority and in section 6, experimental studies are being performed to illustrate the effectiveness of the technique. Concluding remarks are given in the last section.

2. Preliminary work

2.1 Test Suite Management

For a given program P, a testing criterion has to be defined before testing process. Generally speaking, a testing criterion can

be considered as a set of testing requirements. For black box testing, the program specifications are used to identify the testing requirements. For white box testing, the structural components of the program become the testing requirements i.e. statement, branch, D-U pairs etc. For example, if the testing criterion is that each branch of the program be executed at least once, each branch can be considered as a testing requirement. [4].

Let $T=\{t_1,t_2,\ldots t_n\}$ be the initial test suite and $R=\{r_1,r_2,\ldots r_m\}$ be a list of m testing requirements that must be tested to provide the desired testing coverage of the program.

Test cases designed specifically for a requirement may satisfy other requirement. Let $Req(t_i)=\{r_{i1},r_{i2},\ldots r_{ik}\}$ be the set of requirements that test case *ti* satisfies, $R= Req(t_1) \cup Req(t_2) \ldots \cup Req(t_n)$. For each testing requirement *ri*, an associated testing set *Test(ri)* consists of the test cases in T that can be used to test the requirement ri, obviously $T=Test(r1) \cup Test(r2) \ldots \cup Test(rm)$.

These testing requirements and their associated testing sets are used by our code prioritization strategy.

2.2 Req(t_i,t_j)

To a given test case *GTC*, fault localization is needed when the output of the program was wrong. We can preliminarily conclude that the error exists in the requirements *Req(GTC)* it covers.

If *Req(GTC)=1*, undoubtedly the bug exists in the only one requirement, no further work is needed any more. If *Req(GTC)>1*, then we must further localize the exact error requirement.

Definition 1: Let us use *Req(ti,tj)* to stand for the set of common requirements both covered by the test case *ti* and *tj*, i.e., $Req(ti,tj)=Req(ti) \cap Req(tj)$.

Bigger $|Req(GTC,ti)|$ is, more codes in slices of *GTC* and *ti* are common.

The test cases in T are to be allocated into two sets individually: *Right* or *Wrong*. If $Req(GTC,ti) \neq \Phi$, then the successful *ti* will be added in *Right*, while the failed one in *Wrong*.

Then we need to cross out the extra test cases from *Right*. If *Req(GTC)=Req(ti)*, the requirements covered by *GTC* and *ti* are the same, then execution path of *GTC* and *ti* would be the same too, thus *ti* would do nothing to the following work. Cross out *ti* from *Right*. And if *Req(ti)=Req(tj)*, two test cases cover the same requirements and they provide the same information.. When *ti* and *tj* both occur in Right, cross anyone out randomly. So does the *Wrong*.

The test cases in *Right* and *Wrong* will be ordered according to the number of $|Req(GTC,ti)|$, the maximum is the first, the minimum is the last.

3. Sample program

Let's have a sample program to see how to manage the test suite [5], it will be further used to show how to prioritize the code and how to locate the bug in the following sections.

The program in Figure 1 reads the lengths of three sides of a triangle, classifies the triangle, computes its area, and outputs the class and the area computed. Assume that a testing requirement corresponds to the statement coverage.

```
s1: read(a,b,c);
s2: class:=scalene;
s3: if a=b or b=c
s4:   class:=isosceles;
s5: if a=b and b=c
s6:   class:=equilateral;
s7: if a*a=b*b+c*c
s8:  class:=right;
s9: case class of
s10:  right   : area:=b*c/2;
s11:  equilateral : area:=a*2*sqrt(3)/4
s12:  otherwise :  s:=(a+b+c)/2;
s13:          area:=sqrt(s*(s-a)(s-b)(s-c));
s14: end;
      write(class,area);
```

Fig1. Example program

An execution slice with respect to a given test case is the set of code executed by this test. Table 1 gives the test cases in T and its corresponding execution slice. The execution slice of each test case *ti* is namely *Req(ti)*. The program produces correct outputs on all test cases except t5. Because s11 uses the expression a*2 instead of a*a. The result is marked grey in Table1. The associated testing set *Test(ri)* listed in Table 2 can be easily deduced from Table1.

Table 1. The output and requirements satisfied by each test case

Test suite				Req(ti)	Output	
ti	a	b	c	(execution slice of ti)	class	area
t1	2	2	2	1,2,3,4,5,6,7,9,11,14	equilateral	1.73
t2	4	4	3	1,2,3,4,5,7,9,12,14	isosceles	5.56
t3	5	4	3	1,2,3,5,7,8,9,10,14	right	6.00
t4	6	5	4	1,2,3,5,7,9,12,14	scalene	9.92
t5	3	3	3	1,2,3,4,5,6,7,9,11,14	equilateral	2.6
t6	4	3	3	1,2,3,4,5,7,9,12,14	isosceles	4.47

Table 2. Associated testing set of each statement

Statement(ri)	Test(ri)
1	{t1,t2,t3,t4,t5,t6}
2	{t1,t2,t3,t4,t5,t6}

3	{t1,t2,t3,t4,t5,t6}
4	{t1,t2,t5,t6}
5	{t1,t2,t3,t4,t5,t6}
6	{t1,t5}
7	{t1,t2,t3,t4,t5,t6}
8	{t3}
9	{t1,t2,t3,t4,t5,t6}
10	{t3}
11	{t1,t5}
12	{t2,t4,t6}
13	{t2,t4,t6}
14	{t1,t2,t3,t4,t5,t6}

4. Code prioritization methodology

The techniques described in this paper are based on the following observations [5][6][9]:

1) If a statement is not executed under a test case, it cannot affect the program output for that test case.
2) Even if a statement is executed under a test case, it does not necessarily affect the particular output.
3) The likelihood of a piece of code containing a specific fault is proportional to the number of failed tests that execute it.

The likelihood of a piece of code containing a specific fault is inversely proportional to the number of successful tests that execute it.

4.1. The program slicing technique

An execution slice with respect to a given test case is the set of code executed by this test. Based on Observation **1)**, let's have a look at *t5*'s execution slice, of which the statements are bold font in Figure 1. Control didn't reach the statements 8,10,12,13 during execution, we can be sure that the error could not be brought by those statements and must be in its execution slice.

A dynamic slice uses dynamic analysis to identify all and only the statements that contribute to the selected variables of interest on the particular anomalous execution trace. In this way, the size of the slice can be considerably reduced, thus allowing an easier location of the bugs [8]. According to observation **2)**, *t5*'s dynamic slice is depicted grey in Figure 1, even statements 2,3,4,7 are executed under *t5* but do not affect variable area. So the bug must exist in the wrong output *area*'s dynamic slice with respect to *t5*.

To obtain the dynamic slice, the execution history need to be saved firstly and then to recursively traverse the data and control dependence edges in the dynamic dependence graph of the program for a given test case. Although dynamic slice can exactly provide us the statements that do have an effect on the variables of interest, its calculation will exhaust many resources and much time. For inherent exactitude of dynamic slice, it can first of all reduce the searching domain largely comparatively to execution slice.

Let's use E_{gtc} to stand for the dynamic slice of the given test case *GTC*. The bug must in E_{gtc} according to observation **1)** and **2)**. But E_{gtc} may still contain too many codes, and finding out the fault in E_{gtc} is still time-consuming.

4.2. Assistant slice

The execution slice can be directly recorded according to the execution history during the testing course, no extra resources are needed. For further location, we need to introduce much more assistant information. execution slice technique is taken except for *GTC*.

Select out first three test cases from *Right* , the execution slices with respect to them are represented as *E1*, *E2* and *E3* individually. If $Wrong \neq \Phi$, select out the first one from *Wrong* and its corresponding execution slice is expressed as E_f.

4.3. Code prioritization methodology

We first construct the dices as follows: $E_{123}=E1 \cap E2 \cap E3$, $E_{12}=E1 \cap E2$, $E_{1+2+3}=E1 \cup E2 \cup E3$, $E_{1+2}=E1 \cup E2$. Obviously, $E_{123} \subseteq E_{12} \subseteq E1 \subseteq E_{1+2} \subseteq E_{1+2+3}$, the common codes with E_{gtc} are gradually increasing by degrees from E_{123} to E_{1+2+3}.

Definition 2: Let *Prior(X)* be the possibility of containing bug in code segment X.

The rationale behind observation **3)** is that the more successful tests execute a piece of code, the less likely for it to contain any fault. So a requirement in *Req(GTC)* is more satisfied by the test cases in *Right*, less impossible the error code. Thus E_{123} is of most impossibility containing bug. Thus the likelihood of containing bugs for those dices is correspondingly proportional to the number of common codes with E_{gtc}. Obviously $Prior(E_{123}) \leqslant Prior(E_{12}) \leqslant Prior(E1) \leqslant Prior(E_{1+2}) \leqslant Prior(E_{1+2+3})$.

While observation **4)** means the more failed tests execute a piece of code, the more likely for it to contain any fault. We construct a dice $P^0=E_{gtc} \cap E_f$. We can conclude that P^0 is the most suspicious domain of containing error. $Prior(P^0)$ is the highest.

$$Prior(E_{123}) \leqslant Prior(E_{12}) \leqslant Prior(E1) \leqslant Prior(E_{1+2}) \leqslant Prior(E_{1+2+3}) \leqslant Prior(P^0).$$

5. Key algorithms in fault location

We do not know where the bug is before hand because of its randomness. The priority of each dice will be used in this section to concretely localize the fault.

We present two methods to help programmers effectively locate the fault: (1) a refining method to exclude codes from being examined if P^0 contains too many codes. (2) an augmentation method to include additional code based on direct data dependency for inspection if the bug is not in P^0.

5.1. Refining algorithm

If the size of P^0 is small enough, we will directly examine it to see whether the bug is in it. Otherwise, the searching domain will be gradually considered according to the priority.

Suppose the bug is in P^0 at this step.

Based on observation 3), the code that is less likely to contain any fault will be subtracted from P^0 according to the priority in turn. The dices are respectively constructed as follows: $P^5 = P^0 - E_{1+2+3}$, $P^4 = P^0 - E_{1+2}$, $P^3 = P^0 - E1$, $P^2 = P^0 - E_{12}$, $P^1 = P^0 - E_{123}$. Based on this definition, we have $P^0 \supseteq P^1 \supseteq P^2 \supseteq P^3 \supseteq P^4 \supseteq P^5$.

For the size of P^5 is the smallest and the inspecting range is reduced to the most, consider it first. If the bug is not in P^5, maybe too much code is excluded. Then P^4 is the next one to be checked, followed by P^3 and then P^2, P^1 at last P^0.

(1)	k=5
(2)	construct D^k
(3)	examine code in D^k to see whether it contains the bug
(4)	if YES, then STOP
(5)	set k=k−1
(6)	if k<0, then STOP
(7)	go back to (3)

Fig2. Refining algorithm

Let $D^5=P^5$, $D^4=P^4-P^5$, $D^3=P^3-P^4$, $D^2=P^2-P^3$, $D^1=P^1-P^2$ and $D^0=P^0-P^1$. When P^4 is being examined, it is convinced that there are no bugs in P^5. It is clear that we only need to inspect the code in D^4 for $P^4 \supseteq P^5$. Similarly, when examining the dice P^3, P^2, P^1 and P^0 separately, we only need to check $D^k(k=3,2,1,0)$ accordingly.

The refining algorithm is detailed in Figure **2**.

An important point worth noticing is that at the beginning of this section, we assume P^0 containing the bug and then put our focus on how to prioritize code in P^0 so that the bug can be located before all the code in P^0 is examined. However, knowing the location of a bug in advance is not possible. If refining procedure stops at step (4), we have located the bug. Otherwise, the refining procedure stops at step (6) where all the codes in P^0 have been examined.

5.2. Augmenting algorithm

If the bug is not found in the refining phase, we will then look over the remainder of E_{gtc} (i.e., $E_{gtc}-P^0$). Let's use R to represent the codes in $E_{gtc}-P^0$.

Definition 3: There exists a statement $\theta \in R$, it is said to be "**direct data dependency**" relation with P^0 such that $\theta \propto P^0$ if and only if: θ defines a variable x that is used in P^0, or θ uses a variable y defined in P^0. We say that θ is directly data dependent on P^0.

Instead of examining code in R all at one time (i.e., having code in R with the same priority), a better approach is to prioritize the code based on its likelihood of containing the bug in the augment phase. Even the bug is not in P^0, the statement that is directly or indirectly data dependent on P^0 is more suspicious for it to contain any fault than the others in R.

Let's construct the code segment B^1, B^2, B^3,…B^* in-order: If a statement in R is directly data dependent on P^0, adding it into B^1, and B^2 is the union of B^1 and additional code that is directly data dependent on B^1,…B^k is the union of B^{k-1} and additional code that is directly data dependent on B^{k-1}. When no more code can be included into B^k based on direct data dependency, then terminal status is named B^*, that is $B^k = B^{k-1}$. It is evident that $B^1 \subset B^2 \subset B^3 \ldots \subset B^*$.

The augmentation algorithm is listed in Figure **3**.

(1)	k=1, construct $A^1=B^1=\{\theta/\theta \in R \wedge \theta \propto P^0\}$
(2)	examine code in A^k to see whether it contains the bug
(3)	if YES, then STOP
(4)	k=k+1
(5)	$B^k=B^{k-1} \cup \{\theta/\theta \in R \wedge \theta \propto B^{k-1}\}$, $A^k=B^k-B^{k-1}$
(6)	if $B^k=B^{k-1}$ then STOP

Fig3. Augmenting algorithm

In augmenting process, we firstly try to localize the bug form B^1 followed by B^2… till B^*. Additional suspicious code is gradually included that is data dependent on the previous augmented code segment. In each iteration when checking B^k, whose subset B^{k-1} has been inspected and proved having no bug, so the better way is only need to check the code in $A^k = B^k - B^{k-1}$. One exception is the first iteration where B^1 is direct data dependent on P^0.

One important point worth noticing is that if the procedure

stops at step (3), we have successfully located the bug. However, if the procedure stops at (6), we have constructed B^* which still does not contain the bug. In this case, we need to examine the code that is in the failed dynamic slice E_{gtc} but not in B^* nor in P^0 (i.e., code in $E_{gtc} - P^0 - B^*$). Although in theory this is possible, our conjecture is that in practice it does not seem to occur very often. The data listed in Table **4** shows that most P^0 contains the bug.

Input: program P, its test suite *T* and corresponding test requirement set *R*, requirements set *Req(ti)* for each test case ti satisfies, a given failed test case GTC and its dynamic slice E_{gtc}
output: the code segment containing the bug
algorithm:
① for each requirement ri in R, construct associated testing set Test(ri) according to T, R and Req(ti).
② if |Req(GTC)|=1,the only requirement covered by t contains bug, then STOP.
③ for each ti in T, classify it into Right or Wrong in-order according to section 2.2
④ select out first three test cases from Right, obtain the corresponding execution slices E1,E2,E3; If Wrong≠Φ, the execution slice of the first test case in Wrong is E_f.
⑤ $P^0 = E_{gtc} \cap E_f$.
⑥ if the size of P^0 is less than 20% of E_{gtc}, check it directly whether having bug; if found, then STOP.
⑦ fault localization based on refining algorithm, if found, then STOP.
⑧ fault localization based on augmenting algorithm, if found, then STOP.
⑨ fault localization in $E - P^0 - B^*$

Fig 4. Fault localization strategy based on priority

5.3. Incremental algorithm based on priority

Now the whole process to locate the bug based on priority is described in Figure **4**. We prioritize the code in a failed dynamic slice E_{gtc} based on its likelihood of containing bug. The prioritization is done by first using the information of relation between test case and testing requirement obtained from the testing process in Section 4, then refining method in Section 5.1, and finally the augmenting method in Section 5.2. At the worst case, we have to examine all code of E_{gtc}. We will respond to this concern in the next section.

Let's come back to the program in Figure 1. E_{gtc}= *{1,5,6,9,11,14}*, *Right={t2,t3,t4}*, *Wrong*=Φ. Select out the test cases *t2, t3, t4* from *Right*, and their corresponding execution slices are *E1,E2,E3* as depicted in Column 2 of Table 1. Then we can obtain $E_{12}=E_{123}=\{1,2,3,5,7,9,14\}$, $E_{1+2}=E_{1+2+3}=\{1,2,3,5,7,8,9,10,12,14\}$. Thus $P^0=E_{gtc}$, $P^5=P^4=P^3=P^2=P^1=\{6,11\}$, and only two statements left in P^5, we can easily find the error code in s11.

6. Experiment

The choice of faulty programs should represent both the program space and the fault space. In our experiment we introduce six module of Tower Simulator written in C language running Onyx. The test suite is created according to the method in [7]. Black-box test suites are first created, then more test cases are manually added in to ensure that each testing requirement (i.e. statement, branch, D-U pairs and so on) is covered by at least 10 different test cases. A ***version*** of a base module is a mutant by seeding realistic faults into it randomly. We created 1000 faulty versions of these modules by seeding individual faults into the code.

Table 3. Relative data of the experimental program

Program	NoL	|T|	NoV
VisualGen1	516	355	200
VisualGen2	789	378	200
VisualServer1	883	458	200
VisualServer2	897	512	200
ControlPannel	1022	532	100
SimulatorServer	1232	479	100

Table **3** gives the sizes of each base program and its test suite, Column 2 is the number of lines; Column 3 is the size of test suite and Column 4 is the number of program versions we introduced.

Table **4** lists the position of the fault seeded and the phase of fault located in all versions in detail. Column 2 is the number of versions for each base program while Column 3 is the position where fault seeded in the versions: M stands for middle, F front and B bottom. The number of versions for fault located in different phases is presented in Column 3.

From this table, we draw the following conclusion:

- The phase of finding out the fault is irrespective to where the fault is.
- The phase of fault located is concentrated on refining phase⑦. Of 1000 faulty program versions, we have the possibility 75.7% of finding out the bug in refining phase.
- The possibility of P^0 containing bug is very high. The percentage is 80.5%. That means in most cases, the fault can be found even the augmenting method has not yet carried into execution.
- The worst case is that refining and augmenting procedure both failed. We find the fault at phase ⑨. This means the entire dynamic slice *E* has been checked. But it hardly happens practically for the probability is only 0.7%.
- In our experiment, we never succeeded in locating

the bug at phase ②. For our test suite is constructed to ensure every requirement be satisfied by at least 10 test cases.

- **Table 4. N versions of fault located in different phase**

Program	NoV	PoF	Phase of Fault Located				
			②	⑥	⑦	⑧	⑨
1	70	M		2	60	5	1
1	65	F		3	57	4	
1	65	B		3	59	5	
2	65	M		3	49	17	
2	65	F		3	46	16	2
2	70	B		3	45	16	
3	65	M		3	47	12	
3	70	F		3	45	12	
3	65	B		2	45	13	1
4	70	M		3	45	13	
4	65	F		1	43	11	
4	65	B		2	46	12	
5	35	M		2	26	10	1
5	30	F		3	28	11	
5	35	B		3	30	8	1
6	30	M		3	26	10	
6	35	F		2	30	7	
6	35	B		4	30	6	1
	335	M		16	253	67	2
Program	330	F		15	249	61	2
1~6	335	B		17	255	60	3
	1000			48	757	188	7

Table **5** gives us the elaborate information in the refining procedure. The notation *[a,b]* in Column **1** represents the size of E_{gtc} in terms with the percentage between a% and b%. While the notation a/b means there are a dices of present size at present phase and among these a dices there are b dices successfully located the bug. For example, 55/48 in grey in Table 5 means, out of 55 P^0*s*, whose size is less than 20% of E_{gtc}, 48 P^0s have been directly located the bug. Most P^0 is between 30% and 80% of E_{gtc}. So the refining phase is then to be adopted and only 952 P^5*s* over 1000 versions need to be constructed. Most P^5*s*'s size is between 0 to 20% and no P^5 has a size more than 50% of E_{gtc}. At this step, 127 P^5*s* are found out the fault. So next step only 825(1000-48-127) P^4*s* need to be constructed and D^4 is further size reduced. 123 P^4*s* are found out the fault. Similarly only 702 P^3*s* need to be constructed, and so on.

From P^5 to P^0, we only need check very few codes by and large in each step. Totally, we have found out the bug in 805 versions in P^0. At the same time, it implies examining code only in P^0 cannot locate the bugs in 195 versions. Augmenting method is taken.

Table 5. Distribution of dice size and fault located in at refining procedure

Percentage	P^0	D^0	D^1	D^2	D^3	D^4	P^5
[0,20]	55/48	187/13	325/138	492/151	589/118	733/94	893/107
[20,30]	75/-	21/-	27/6	47/36	113/45	55/21	36/17
[30,50]	305/-	-	-	-	-/-	37/8	23/3
[50,80]	503/-	-	-	-	-	-	-
[80,100]	62/-	-	-	-	-	-	-
Total	1000/48	208/13	352/144	539/187	702/163	825/123	952/127

Table 6 presents the relative data in augmenting procedure. **Good A^k** means the fault is discovered at dice A^k. When inspecting the code in A^k, whereas B^k is indeed checked if k>1. According to the direct data dependent relation, priority is evaluated to the code in *R*. Basically A^4 is the worst case that no more codes could be added in. Most A^k(or B^k) has a size less than 20% of E_{gtc}. If refining procedure failed, the augmenting method really works.

Table 6. Size distribution of good A^ks at augmenting procedure

Percentage	A^0	B^2	B^3	B^4	$A^*(k\geqslant 5)$
[0,10]	45	53	13	7	-
[10,20]	37	6	5	2	-
[20,30]	9	-	-	-	-
[30,100]	-	-	-	-	-
Total	91	59	18	9	

To conclude, our data indicates that the effectiveness of fault location. The size of dice being checked in each step is very small. At most cases, P^0 contains the bug and augmenting procedure is not needed. Fortunately, we found that only 0.8% failed to locate the fault after refining and augmenting. Our incremental strategy is independent of the fault type, fault position and good expert knowledge of the program being debugged. It can be automated.

7. Conclusion

To a given test case *GTC*, fault location has to be proceeded when the output of a program is wrong. We first make full use of the information provided by testing process, three successful test cases and one more failed test case are selected out. Then the code is prioritized by the likelihood of containing bug. Some empirical observations and heuristic approach are combined with program slice technique to prioritize the code. The rational behind this is that the more successful tests execute a piece of code, the less likely for it to contain any fault, vice versa.

We start with *GTC*'s dynamic slice, which only consists of the code influencing the wrong output variable. P^0 is constructed with highest priority by separating E_f from E_{gtc}. Then we follow

the refining procedure discussed in Section 5.1 to construct dices P^5, P^4, P^3, P^2 and P^1 by subtracting the code from P^0 based on priority gradually. We will check the dices from P^5 to P^0 in turn. If we cannot find the bug in P^0, it implies we need to examine the code in *R*. In this case, we follow the augmenting procedure discussed in Section 5.2 to first construct B^1 that is directly data dependent on P^0. Even the bug is not in P^0, the statement that is directly or indirectly data dependent on P^0 is more suspicious for it to contain any fault than the others in *R*. Code prioritization at this phase is implemented according to the direct data dependent relation, thus B^1, B^2,... B^* are constructed and checked in-order. If the bug is still not found, we then inspect the last piece of code in $E_{gtc} - P^0 - B^*$.

In short, we propose an incremental fault location strategy based on the code priority, which is correspondent with the likelihood of containing bug. The most suspicious code is examined first, and then step-by-step to increase the searching domain by including additional code based on the priority.

We conducted an experiment to show the effectiveness of our fault localization method. In most cases, the bug can be found in the refining phase or even before. The probability is 80.5%. The worst situation is equal to examine the whole dynamic slice of given test case *GTC* when both refining and augmenting method are failed. But the chance for this is only 0.8%.

An interesting future study is to compare the effectiveness when adopting different slice techniques. For example, if *E1*, *E2*, *E3* and E_f are all computed by dynamic slice technique, could the phase of fault location be brought forward? Even the phase were moved forward, could time and money be spared comparative to calculate the dynamic slices. Another interesting future work is to apply our method to industry projects to examine how much time programmers can save by using our method in locating bugs in comparison to other approaches.

References

[1]M Weiser. Programmers Use Slices When Debugging [J].Communications of the ACM, 1982, 25(7):46-452.

[2]Li Bixin. Program Slice Technique and its Application in Object-Oriented Software Metrics and Software Test[D]. Dissertation for PhD, School of Computer Software and Theory, Nanjing University, 2000.11: pp.62-67

[3]H. Agrawal, J. R. Horgan, S. London, et al. Fault Localization Using Execution Slices and Dataflow Tests. Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, October 1995, pp:143-151.

[4] M.J. Harrold, R. Gupta, M.L. Soffa. A Methodology for Controlling the Size of A Test Suite, ACM Transaction on Software Engineering and Methodology, July(1993)270-285.

[5]N. Mansour, R. Bahsoon. Reduction-based methods and metrics for selective regression testing. Information and Software Technology, 44(2002)431-443

[6]H. Agrawal, J.R. Horgan, E.W. Krauser, Incremental regression testing. Proceeding of the Conference on Software Maintaiance, 1993,pp.299-308.

[7]M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and control flow-based test adequacy criterions. In Proc. of the 16th Int'l. Conf. on software. Eng., pp. 191-200, May 1994.

[8] H. Agrawal, R. A. DeMillo, and E. H. Spafford, Dubugging with Dynamic Slicing and Backtracking, Software-Practice & Experience, 23(6):589-616, June, 1996

[9] W. E. Wong, T. Sugeta, Y. Qi, et al., Smart Debugging Software Architecture Design in SDL, in Proceedings of the 27th IEEE International Computer Software and Applications Conference, November 2003. pp. 41-47.