Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

The Journal of Systems and Software 79 (2006) 891–903

The Journal of Systems and Software

www.elsevier.com/locate/jss

Effective program debugging based on execution slices and inter-block data dependency

W. Eric Wong [*], Yu Qi

Department of Computer Science, University of Texas at Dallas, MS EC 31, 2601 N. Floyd Road, Richardson, TX 75083, United States

Received 6 December 2004; accepted 20 June 2005
Available online 30 January 2006

Abstract

Localizing a fault in program debugging is a complex and time-consuming process. In this paper we present a novel approach using execution slices and inter-block data dependency to effectively identify the locations of program faults. An execution slice with respect to a given test case is the set of code executed by this test, and two blocks are data dependent if one block contains a definition that is used by another block or vice versa. Not only can our approach reduce the search domain for program debugging, but also prioritize suspicious locations in the reduced domain based on their likelihood of containing faults. More specifically, we use multiple execution slices to prioritize the likelihood of a piece of code containing a specific fault. In addition, the likelihood also depends on whether this piece of code is data dependent on other suspicious code. A debugging tool, DES*i*D, was developed to support our method. A case study that shows the effectiveness of our method in locating faults on an application developed for the European Space Agency is also reported.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Software testing; Fault localization; Program debugging; Execution slice; Inter-block data dependency; DES*i*D

1. Introduction

No matter how much effort is spent on testing a program,[1] it appears to be a fact of life that software defects are introduced and removed continually during software development processes. To improve the quality of a program, we have to remove as many defects in the program as possible without introducing new bugs at the same time. However, localizing a fault in program debugging can be a complex and time-consuming process. One intuitive way to debug a program when it shows some abnormal behavior is to analyze its memory dump. Another way is to insert *print* statements around suspicious code to print out the values of some variables. While the former is not often used now because it might require an analysis of a tremendous amount of data, the latter is still used. However, users need to have a good understanding of how a program is executed with respect to a given test case that causes the trouble and then insert only the necessary (i.e., neither too many nor too few) print statements at the appropriate locations. As a result, it is also not an ideal debugging technique for identifying the locations of faults.

To overcome the problems discussed above, debugging tools such as DBX and Microsoft VC++ debugger have been developed. They allow users to set break points along a program execution and examine values of variables as well as internal states at each break point, if so desired. One can think of this approach as inserting print statements into the program except that it does not require the physical insertion of any print statements. Break points can be pre-set before the execution or dynamically set on the fly by the user in an interactive way during program execution. Two types of execution are available: a continuous execution from one break point to the next break

[*] Corresponding author. Tel.: +1 972 883 6619; fax: +1 972 883 2349.
E-mail address: ewong@utdallas.edu (W.E. Wong).
URL: http://www.utdallas.edu/~ewong (W.E. Wong).

[1] In this paper, we use "bugs", "faults" and "software defects" interchangeably. We also use "program", "application" and "software" interchangeably. In addition, "locating a bug" and "localizing a fault" have the same meaning.

point, or a stepwise execution starting from a break point. While using these tools, users must first decide where the break points should be. When the execution reaches a break point, they are allowed to examine the current program state and to determine whether the execution is still correct. If not, the execution should be stopped, and some analysis as well as backtracking may be necessary in order to locate the fault(s). Otherwise, the execution goes on either in a continuous mode or in a stepwise mode. In summary, these tools provide a snapshot of the program state at various break points along an execution path. One major disadvantage of this approach is that it requires users to develop their own strategies to avoid examining too much information for nothing. Another significant disadvantage is that it cannot reduce the search domain by prioritizing code based on the likelihood of containing faults on a given execution path.

To develop a more effective debugging strategy, program slicing has been used. In this paper, we propose a method based on execution slices and inter-block data dependency. For a given test case, its execution slice is the set of code (basic blocks[2] in our case) executed by this test. An execution slice-based technique as reported in (Agrawal et al., 1995) can be effective in locating some program bugs, but not for others, especially those in the code that are executed by both the failed and the successful tests. Another problem with this technique 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 still be too much code that needs to be examined. We propose an augmentation method and a refining method to overcome these problems. The former includes additional code in the search domain for inspection based on its inter-block data dependency with the code which is currently being examined, whereas the latter excludes less suspicious code from the search domain using the execution slices of additional successful tests. Stated differently, the augmentation and refining methods are used to help programmers better prioritize code based on its likelihood of containing program bug(s) for an effective fault localization. A case study was conducted to demonstrate the effectiveness of our method, and a tool, DES*i*D (which stands for a Debugging tool based on Execution Slices and inter-block data Dependency) was implemented to automate the slice generation, code prioritization, and data collection.

The rest of the paper is organized as follows. Section 2 describes our method. Section 3 explains our method with an example, and Section 4 reports a case study. Section 5 discusses issues related to our method. Section 6 presents an overview of some related studies. Finally, in Section 7 we offer our conclusions and recommendations for future research.

[2] A basic block, also known as a block, is a sequence of consecutive statements or expressions containing no transfers of control except at the end, so that if one element of it is executed, all are. This of course assumes that the underlying hardware does not fail during the execution of a block. Hereafter, we refer to a basic block simply as a block.

2. Methodology

Suppose for a given program bug, we have identified one failed test (say F) and one successful test (say S). Suppose also their execution slices are denoted by E_F and E_S, respectively. We can construct a dice $\mathscr{D}^{(1)} = E_\mathrm{F} - E_\mathrm{S}$. We present two methods to help programmers effectively locate this bug: an augmentation method to include additional code for inspection if the bug is not in $\mathscr{D}^{(1)}$, and a refining method to exclude code from being examined if $\mathscr{D}^{(1)}$ contains too much code.

2.1. Augmenting bad $\mathscr{D}^{(1)}$

If the bug is not in $\mathscr{D}^{(1)}$, we need to examine additional code from the *rest* of the failed execution slice (i.e., code from $E_\mathrm{F} - \mathscr{D}^{(1)}$). Let us use Φ to represent the set of code that is in $E_\mathrm{F} - \mathscr{D}^{(1)}$. For a block β, the notation $\beta \in \Phi$ implies β is in the failed execution slice E_F but not in $\mathscr{D}^{(1)}$. We also define a "direct data dependency" relation Δ such that $\beta\Delta\mathscr{D}^{(1)}$ if and only if β defines a variable x that is used in $\mathscr{D}^{(1)}$ or β uses a variable y defined in $\mathscr{D}^{(1)}$. We say that β is directly data dependent on $\mathscr{D}^{(1)}$.

Instead of examining code in Φ all at one time (i.e., having code in Φ with the same priority), a better approach is to prioritize such code based on its likelihood of containing the bug. To do so, we use the augmentation procedures outlined as follows:

```
A_1: Construct $\mathscr{A}^{(1)}$, the augmented code segment from the first iteration, such that $\mathscr{A}^{(1)} = \{\beta | \beta \in \Phi \wedge (\beta\Delta\mathscr{D}^{(1)})\}$.
A_2: Set $k = 1$.
A_3: Examine code in $\mathscr{A}^{(k)}$ to see whether it contains the bug.
A_4: If YES, then STOP because we have located the bug.
A_5: Set $k = k + 1$.
A_6: Construct $\mathscr{A}^{(k)}$, the augmented code segment from the $k$th iteration such that $\mathscr{A}^{(k)} = \mathscr{A}^{(k-1)} \cup \{\beta | \beta \in \Phi \wedge (\beta\Delta\mathscr{A}^{(k-1)})\}$.
A_7: If $\mathscr{A}^{(k)} = \mathscr{A}^{(k-1)}$ (i.e., no new code can be included from the $(k-1)$th iteration to the $k$th iteration), then STOP. At this point we have $\mathscr{A}^{(*)}$, the final augmented code segment, equals $\mathscr{A}^{(k)}$ (and $\mathscr{A}^{(k-1)}$ as well).
A_8: Go back to step A_3.
```

Stated differently, $\mathscr{A}^{(1)}$ contains the code that is directly data dependent on $\mathscr{D}^{(1)}$, and $\mathscr{A}^{(2)}$ is the union of $\mathscr{A}^{(1)}$ and additional code that is directly data dependent on $\mathscr{A}^{(1)}$. It is clear that $\mathscr{A}^{(1)} \subset \mathscr{A}^{(2)} \subset \mathscr{A}^{(3)} \cdots \subset \mathscr{A}^{(*)}$.

Our incremental approach is that if a program bug is not in $\mathscr{D}^{(1)}$, we will then try to locate it starting from $\mathscr{A}^{(1)}$ followed by $\mathscr{A}^{(2)}$ and so on until $\mathscr{A}^{(*)}$. For each iteration we include additional code that is data dependent on the previous augmented code segment. One exception is the first iteration where $\mathscr{A}^{(1)}$ is data dependent on $\mathscr{D}^{(1)}$. The rationale behind this augmentation method is that even though a program bug is not in the suspicious area (such as $\mathscr{D}^{(1)}$), it is very likely that it has something to do with the code that is directly data dependent on the code in $\mathscr{D}^{(1)}$ or some $\mathscr{A}^{(k)}$ constructed at step A_6. An example of this appears in Section 3. In addition, the results of our case study on a SPACE application (refer to Section 4.3.1) also indicate that our augmentation method is an effective approach to include additional code for locating a program bug when it is not in $\mathscr{D}^{(1)}$.

One important point worth noting is that if the procedure stops at step A_4, we have successfully located the bug. However, if the procedure stops at A_7, we have constructed $\mathscr{A}^{(*)}$ which still does not contain the bug. In this case, we need to examine the code that is in the failed execution slice (E_F) but not in $\mathscr{A}^{(*)}$ nor in $\mathscr{D}^{(1)}$ (i.e., code in $E_F - \mathscr{D}^{(1)} - \mathscr{A}^{(*)}$). 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 2 show that most $\mathscr{A}^{(*)}$'s are good, i.e., containing the bug. We understand more empirical data need to be collected to verify this statement. However, even if the bug is in $E_F - \mathscr{D}^{(1)} - \mathscr{A}^{(*)}$, we at most examine the entire failed execution slice (i.e., every piece of code in E_F). This implies that even in the worst case our augmentation method does not require a programmer to examine any extra code that is not in E_F.

2.2. Refining good $\mathscr{D}^{(1)}$

Suppose a program bug is in $\mathscr{D}^{(1)}$; we may still end up with too much code that needs to be examined. In other words, $\mathscr{D}^{(1)}$ still contains too much code. If that is the case, we can further prioritize code in this good $\mathscr{D}^{(1)}$ by using additional successful tests. More specifically, we can use the refining procedures outlined as follows:

R_1: Randomly select additional k distinct successful tests (say $t_1, t_2, \ldots, t_k$).
R_2: Construct the corresponding execution slices (denoted by $\Theta_1, \Theta_2, \ldots, \Theta_k$).
R_3: Set $m = k$.
R_4: Construct $\mathscr{D}^{(m+1)}$ such that $\mathscr{D}^{(m+1)} = \{\beta | \beta \in \mathscr{D}^{(1)} \wedge \beta \notin \Theta_n \text{ for } n = 1, 2, \ldots, m\}$, that is, $\mathscr{D}^{(m+1)} = \mathscr{D}^{(1)} - \cup \{\Theta_n \text{ for } n = 1, 2, \ldots, m\}$.
R_5: Examine code in $\mathscr{D}^{(m+1)}$ to see whether it contains the bug.
R_6: If YES, then STOP because we have located the bug.
R_7: Set $m = m - 1$.
R_8: If $m = 0$, then STOP and examine code in $\mathscr{D}^{(1)}$.
R_9: Go back to step R_4.

Stated differently, $\mathscr{D}^{(2)} = \mathscr{D}^{(1)} - \Theta_1 = E_F - E_S - \Theta_1$, $\mathscr{D}^{(3)} = \mathscr{D}^{(1)} - \Theta_1 - \Theta_2 = E_F - E_S - \Theta_1 - \Theta_2$, etc. This implies $\mathscr{D}^{(2)}$ is constructed by subtracting the union of the execution slices of two successful tests (E_S and Θ_1 in our case) from the execution slice of the failed test (E_F in our case), whereas $\mathscr{D}^{(3)}$ is obtained by subtracting the union of the execution slices of three successful tests (namely, E_S, Θ_1 and Θ_2) from E_F. Based on this definition, we have $\mathscr{D}^{(1)} \supseteq \mathscr{D}^{(2)} \supseteq \mathscr{D}^{(3)}$, etc.

For explanatory purposes, assume $k = 2$. Our refining method suggests that we can reduce the code to be examined by first inspecting code in $\mathscr{D}^{(3)}$ followed by $\mathscr{D}^{(2)}$ and then $\mathscr{D}^{(1)}$. The rationale behind this is that if a piece of code is executed by some successful tests, then it is less likely to contain any fault. In this example, if $\mathscr{D}^{(3)}$ or $\mathscr{D}^{(2)}$ contains the bug, the refining procedure stops at step R_6. Otherwise, we have the bug in $\mathscr{D}^{(1)}$, but not in $\mathscr{D}^{(2)}$ nor in $\mathscr{D}^{(3)}$. If so, the refining procedure stops at step R_8 where all the code in $\mathscr{D}^{(1)}$ will be examined.

An important point worth noting is that at the beginning of this section, we assume $\mathscr{D}^{(1)}$ contains the bug and then put our focus on how to prioritize code in $\mathscr{D}^{(1)}$ so that the bug can be located before all the code in $\mathscr{D}^{(1)}$ is examined. However, knowing the location of a program bug in advance is not possible except for a controlled experiment such as the one reported in Section 4. We will respond to this concern in the next section.

2.3. An incremental debugging strategy

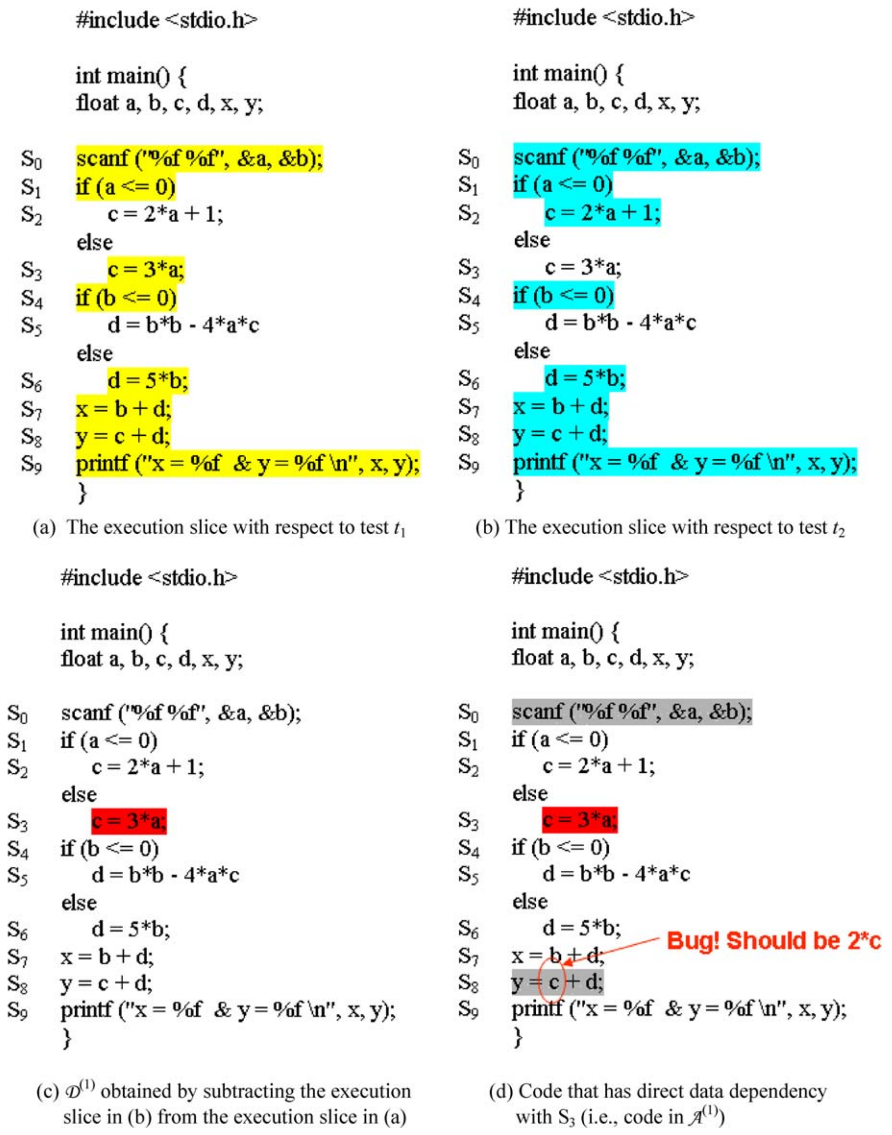
To effectively locate a bug in a program requires a good expert knowledge of the program being debugged. If a programmer has a strong feeling about where a bug might be, (s)he should definitely examine that part of the program. However, if this does not reveal the bug, it is better to use a systematic approach by adopting some heuristics supported by good reasoning and proven to be effective in some case studies, rather than take an ad hoc approach without any intuitive support, to find the location of the bug.

For explanatory purposes, let us assume we have one failed test and three successful tests. We suggest an incremental debugging strategy by first examining the code in $\mathscr{D}^{(3)}$ followed by $\mathscr{D}^{(2)}$ and then $\mathscr{D}^{(1)}$. This is based on the assumption that the bug is in the code which is executed by the failed test but not the successful test(s). If this assumption does not hold (i.e., the bug is not in $\mathscr{D}^{(1)}$), then we need to inspect additional code in E_F (the failed execution slice) but not in $\mathscr{D}^{(1)}$. We can follow the augmentation procedure discussed earlier to start with code in $\mathscr{A}^{(1)}$, followed by $\mathscr{A}^{(2)}, \ldots,$ up to $\mathscr{A}^{(*)}$. If $\mathscr{A}^{(*)}$ does not contain the bug, then we need to examine the last piece of code, namely, $E_F - \mathscr{D}^{(1)} - \mathscr{A}^{(*)}$. In short, we prioritize code in a failed execution slice based on its likelihood of containing the bug. The prioritization is done by first using the refining method in Section 2.2 and then the augmentation method in Section 2.1. In the worst case, we have to examine all the code in the failed execution slice. More discussions can be found in Section 5.

3. An example

In this section, we use an example to explain in detail how our method can be used to prioritize suspicious code in a program for effectively locating a bug. Without resorting to complex semantics, we use a program P which takes two inputs a and b and computes two outputs x and y as shown in Fig. 1. Suppose P has a bug at S_8 and the correct statement should be "$y = 2 * c + d$" instead of "$y = c + d$". The execution of P on a test t_1 where $a = 1$ and $b = 2$ fails because y should be 16 instead of 13. However, the execution of P on another test t_2 where $a = -0.5$ and $b = 3$ does not cause any problem. This is because when c is zero (computed at S_2) the bug is masked (i.e., there is no difference between "$2 * c + d$" and "$c + d$" when c is zero). As a result, we have found one failed test t_1 and one successful test t_2.

The execution slices for t_1 (E_1) and t_2 (E_2) are displayed in parts (a) and (b) of Fig. 1. Part (c) gives $\mathscr{D}^{(1)}$ obtained by subtracting E_2 from E_1. It contains only one statement S_3. Our first step is to examine the code in $\mathscr{D}^{(1)}$ which, unfortunately, does not contain the bug. Next, rather than trying to locate the bug from the code which is in E_1 (the *failed* execution slice) but not in $\mathscr{D}^{(1)}$ (S_3 in our case), a better approach is to prioritize such code based on its likelihood of containing the bug. Stated differently, instead of having S_0, S_1, S_4, S_6, S_7, S_8 and S_9 with the same priority, we give a higher priority to code that has data dependency with S_3. As shown in part (d), $\mathscr{A}^{(1)}$ (the augmented code after the first iteration) contains additional code at S_0 and S_8. This is because the variable a used at S_3 is defined at S_0 through a *scanf* statement, and the variable c defined at S_3 is used at S_8. This also implies S_0 and S_8 should have a higher priority and should be examined next. Since S_8 is part of the code to



```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(a) The execution slice with respect to test t_1

```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(b) The execution slice with respect to test t_2

```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(c) $\mathscr{D}^{(1)}$ obtained by subtracting the execution slice in (b) from the execution slice in (a)

```
     #include <stdio.h>

     int main() {
     float a, b, c, d, x, y;

S0   scanf ("%f %f", &a, &b);
S1   if (a <= 0)
S2       c = 2*a + 1;
     else
S3       c = 3*a;
S4   if (b <= 0)
S5       d = b*b - 4*a*c
     else
S6       d = 5*b;
S7   x = b + d;
S8   y = c + d;
S9   printf ("x = %f  & y = %f \n", x, y);
     }
```

(d) Code that has direct data dependency with S_3 (i.e., code in $\mathscr{A}^{(1)}$)

Fig. 1. Two execution slices, the corresponding $\mathscr{D}^{(1)}$ and $\mathscr{A}^{(1)}$ for the example discussed in Section 3. The reader is referred to the web version of this article for a display in color.

be examined next, we conclude that we have successfully located the bug.

Note that $\mathscr{A}^{(2)}$ (the augmented code after the second iteration) contains additional code at S_1, S_4, S_6, S_7 and S_9 because S_1 uses the variable a defined at S_0, and S_4, S_6 and S_7 use the variable b defined at S_0, and S_9 uses the variable y defined at S_8. Hence, although S_1, S_4, S_6, S_7 and S_9 are not included in $\mathscr{A}^{(1)}$, they will be eventually in $\mathscr{A}^{(*)}$ (or more precisely $\mathscr{A}^{(2)}$ in this case). However, there is no need to go through more than one iteration in this example, i.e., no need to examine S_1, S_4, S_6, S_7 and S_9.

4. A case study

We first provide an overview of SPACE and DES*i*D, the target program and the tool used in this study, followed by a description of test case generation, fault set, and erroneous program preparation. Then, we present our results and analysis based on these results.

4.1. Program and tool

SPACE provides a language-oriented user interface that allows the user to describe the configuration of an array of antennas by using a high level language (Cancellieri and Giorgi, 1994). Its purpose is to prepare a data file in accordance with a predefined format and characteristics from a user, given the array antenna configuration described in a language-like form. The application consists of about 10,000 lines of code.

A fundamental problem with applying the method discussed in Section 2 for debugging C code[3] is that we need to have a tool to support this process. It is impossible to manually collect the execution slices and prioritize code in these slices, even for a small C program, because it can be very time consuming and mistake prone. With this in mind, we developed DES*i*D on top of ATAC, which is part of the Telcordia Software Visualization and Analysis Toolsuite (Horgan and London, 1991; χSuds User's Manual, 1998).[4] Given a C program, ATAC instruments it at compile time by inserting probes at appropriate locations and builds an executable based on the instrumented code. For each program execution, ATAC saves the traceability, including how many times each block is executed by a test case, in a trace file. With this information we can use DES*i*D to compute the execution slice for each test in terms of a set of blocks. In addition, we can use DES*i*D to compute the corresponding $\mathscr{D}^{(1)}$, $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ as well as $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots, \mathscr{A}^{(*)}$ for locating program bugs.

DES*i*D provides two different user interfaces: a graphical user interface and a command-line text interface. The former allows programmers to select a failed test and up to three different successful tests through a user friendly graphical interface. In addition, it can also highlight code in the corresponding failed execution slice, and $\mathscr{D}^{(1)}, \mathscr{D}^{(2)}, \mathscr{D}^{(3)}, \mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots,$ as well as $\mathscr{A}^{(*)}$ with appropriate background colors,[5] which allows programmers to easily visualize the part(s) of a program that should be examined first for locating a bug. In Fig. 2 part of the SPACE program is highlighted in red based on whether it is in $\mathscr{D}^{(1)}$, $\mathscr{D}^{(2)}$ or $\mathscr{D}^{(3)}$. Part (a) shows three pieces of code in $\mathscr{D}^{(1)}$, part (b) shows two pieces in $\mathscr{D}^{(2)}$, and part (c) shows only one line of code in $\mathscr{D}^{(3)}$. Clearly, $\mathscr{D}^{(1)}$ subsumes $\mathscr{D}^{(2)}$ which then subsumes $\mathscr{D}^{(3)}$. On the other hand, code can be highlighted in blue based on whether it is in $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots,$ or $\mathscr{A}^{(*)}$. Parts (b) and (c) of Fig. 3 show such code with respect to $\mathscr{A}^{(1)}$ and $\mathscr{A}^{(2)}$, respectively. We have $\mathscr{A}^{(1)}$ as a subset of $\mathscr{A}^{(2)}$. As explained in Section 2, it is up to the programmers to decide whether they should start with $\mathscr{D}^{(3)}$, $\mathscr{D}^{(2)}$ or $\mathscr{D}^{(1)}$. If the bug is not in $\mathscr{D}^{(1)}$, then they need to examine additional code in $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}$, etc.

While the graphical user interface is useful to help developers find the location of a bug which they do not know in advance, it is not convenient (because code visualization through such an interface cannot be automated) for a controlled experiment like the one reported here. This is because in the latter the fault locations are already known before the experiment, and the objective is not to *find* the location of a bug but to *verify* whether our method can help programmers effectively locate a bug, that is, whether the bug is in $\mathscr{D}^{(1)}$, or $\mathscr{D}^{(2)}$, or $\mathscr{D}^{(3)}$, or $\mathscr{A}^{(1)}$, or $\mathscr{A}^{(2)}$, etc. To overcome this problem, DES*i*D provides a command-line text interface in which appropriate routines can be invoked directly to compute various dices and augmented code based on a given failed test and some successful tests. Moreover, such commands which take as input the failed and successful tests can be saved in a batch file for automatic execution. It is this approach which makes DES*i*D very robust for our case study.

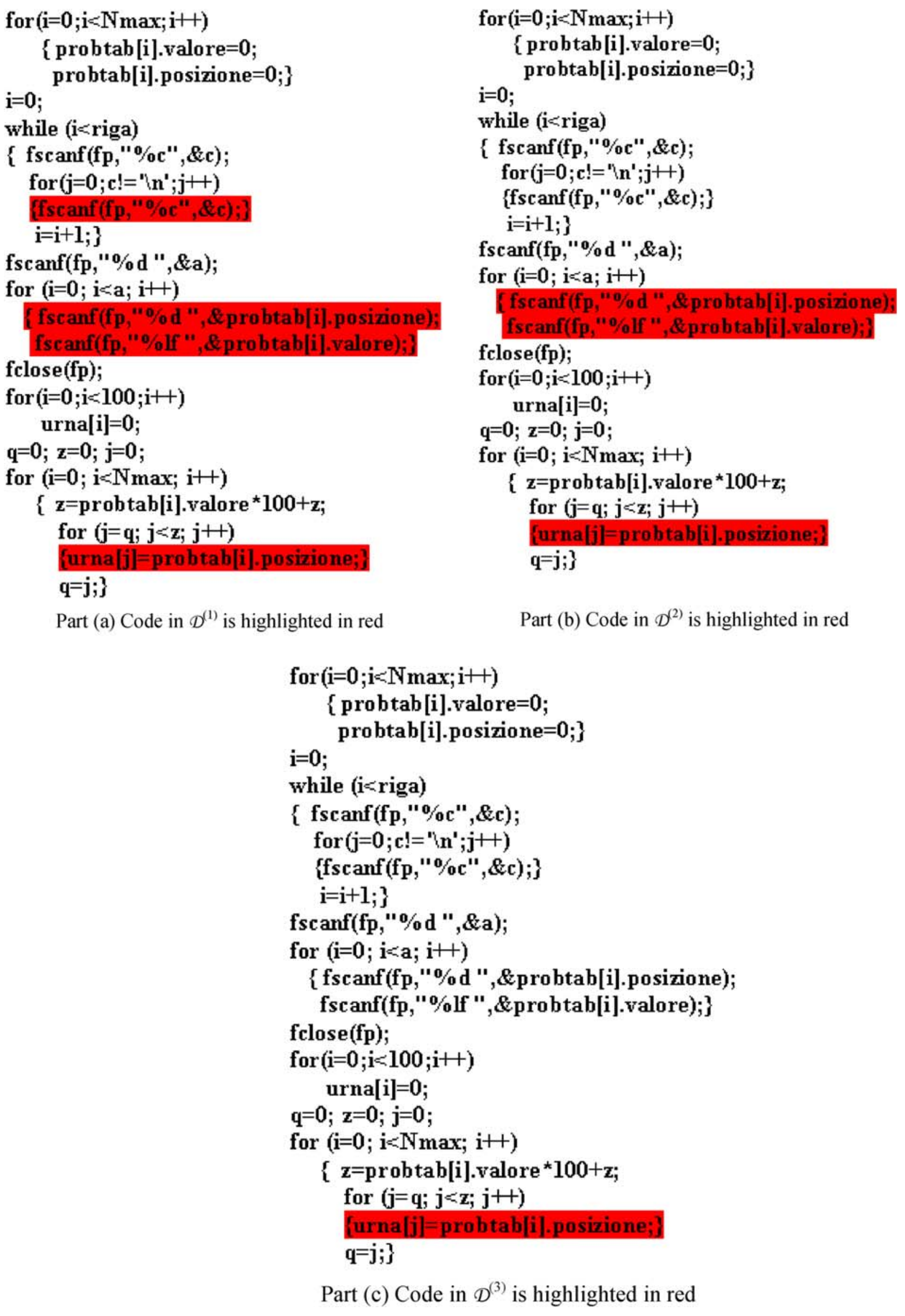
4.2. Test case generation, fault set, and erroneous program preparation

An operational profile, as formalized by Musa (1993) and used in our experiment, is a set of the occurrence probabilities of various software functions. To obtain an operational profile for SPACE we identified the possible functions of the program and generated a graph capturing the connectivity of these functions. Each node in the graph represents a function. Two nodes, A and B, are connected

[3] The method discussed in Section 2 also applies to other programming languages such as C++ and Java. For experiments and the associated tool development, however, we have to select a specific language—C in our case.

[4] The tool χSlice (a slicing tool used in our previous study, Agrawal et al., 1995) can be viewed as a preliminary version of DES*i*D with limited features. More specifically, χSlice can only compute code in $\mathscr{D}^{(1)}$ but not in $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$. Neither can χSlice compute code in $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots, \mathscr{A}^{(*)}$. As a result, χSlice does not work if a program bug is not located in $\mathscr{D}^{(1)}$.

[5] For interpretation of color in all the figures, the reader is referred to the web version of this article.



```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
     urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
    { z=probtab[i].valore*100+z;
       for (j=q; j<z; j++)
       {urna[j]=probtab[i].posizione;}
       q=j;}
```

Part (a) Code in $\mathcal{D}^{(1)}$ is highlighted in red

```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
     urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
    { z=probtab[i].valore*100+z;
       for (j=q; j<z; j++)
       {urna[j]=probtab[i].posizione;}
       q=j;}
```

Part (b) Code in $\mathcal{D}^{(2)}$ is highlighted in red

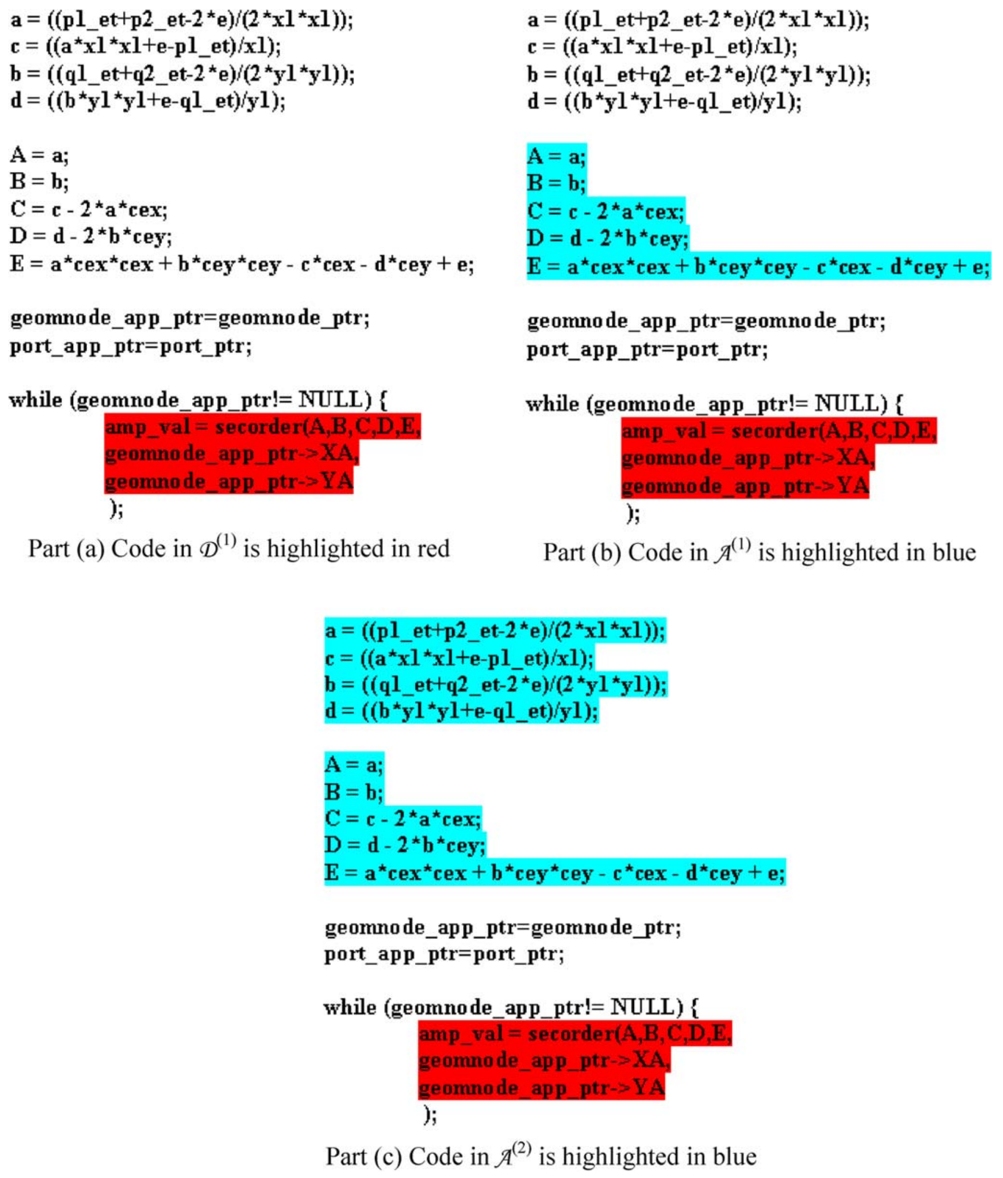
```
for(i=0;i<Nmax;i++)
    { probtab[i].valore=0;
      probtab[i].posizione=0;}
i=0;
while (i<riga)
{ fscanf(fp,"%c",&c);
   for(j=0;c!='\n';j++)
   {fscanf(fp,"%c",&c);}
    i=i+1;}
fscanf(fp,"%d ",&a);
for (i=0; i<a; i++)
  { fscanf(fp,"%d ",&probtab[i].posizione);
   fscanf(fp,"%lf ",&probtab[i].valore);}
fclose(fp);
for(i=0;i<100;i++)
     urna[i]=0;
q=0; z=0; j=0;
for (i=0; i<Nmax; i++)
    { z=probtab[i].valore*100+z;
       for (j=q; j<z; j++)
       {urna[j]=probtab[i].posizione;}
       q=j;}
```

Part (c) Code in $\mathcal{D}^{(3)}$ is highlighted in red

Fig. 2. Highlighting of code based on $\mathcal{D}^{(1)}$, $\mathcal{D}^{(2)}$ and $\mathcal{D}^{(3)}$.

if control can flow from function A to function B. There is a unique start and end node representing functions at which execution begins and terminates, respectively. A path through the graph from the start node to the end node represents one possible program execution. To estimate the occurrence probability of the software functions, each arc is assigned a transition probability, i.e., the probability of control flowing between the nodes connected by the arc. For example, if node A is connected to nodes B, C and D, and the probabilities associated with arcs A–B, A–C and A–D are, respectively, 0.3, 0.6 and 0.1, then after the execution of function A the program will perform functions B, C or D, respectively, with probability 0.3, 0.6 and 0.1. Transition probabilities were determined by running a set of representative test cases. Based on this profile, a pool of 1000 distinct test cases was created.

The fault set for the program used in this study is obtained from the error-log maintained during its testing and integration phase. Ten faults were used in our study. For convenience, each fault is numbered as F_k where k is from 01 to 10. They were injected one at a time, i.e., one erroneous program was created for each fault.



```
a = ((p1_et+p2_et-2*e)/(2*x1*x1));
c = ((a*x1*x1+e-p1_et)/x1);
b = ((q1_et+q2_et-2*e)/(2*y1*y1));
d = ((b*y1*y1+e-q1_et)/y1);

A = a;
B = b;
C = c - 2*a*cex;
D = d - 2*b*cey;
E = a*cex*cex + b*cey*cey - c*cex - d*cey + e;

geomnode_app_ptr=geomnode_ptr;
port_app_ptr=port_ptr;

while (geomnode_app_ptr!= NULL) {
            amp_val = secorder(A,B,C,D,E,
            geomnode_app_ptr->XA,
            geomnode_app_ptr->YA
            );
```

Part (a) Code in $\mathscr{D}^{(1)}$ is highlighted in red

```
a = ((p1_et+p2_et-2*e)/(2*x1*x1));
c = ((a*x1*x1+e-p1_et)/x1);
b = ((q1_et+q2_et-2*e)/(2*y1*y1));
d = ((b*y1*y1+e-q1_et)/y1);

A = a;
B = b;
C = c - 2*a*cex;
D = d - 2*b*cey;
E = a*cex*cex + b*cey*cey - c*cex - d*cey + e;

geomnode_app_ptr=geomnode_ptr;
port_app_ptr=port_ptr;

while (geomnode_app_ptr!= NULL) {
            amp_val = secorder(A,B,C,D,E,
            geomnode_app_ptr->XA,
            geomnode_app_ptr->YA
            );
```

Part (b) Code in $\mathscr{A}^{(1)}$ is highlighted in blue

```
a = ((p1_et+p2_et-2*e)/(2*x1*x1));
c = ((a*x1*x1+e-p1_et)/x1);
b = ((q1_et+q2_et-2*e)/(2*y1*y1));
d = ((b*y1*y1+e-q1_et)/y1);

A = a;
B = b;
C = c - 2*a*cex;
D = d - 2*b*cey;
E = a*cex*cex + b*cey*cey - c*cex - d*cey + e;

geomnode_app_ptr=geomnode_ptr;
port_app_ptr=port_ptr;

while (geomnode_app_ptr!= NULL) {
            amp_val = secorder(A,B,C,D,E,
            geomnode_app_ptr->XA,
            geomnode_app_ptr->YA
            );
```

Part (c) Code in $\mathscr{A}^{(2)}$ is highlighted in blue

Fig. 3. Highlighting of code based on $\mathscr{D}^{(1)}$, $\mathscr{A}^{(1)}$ and $\mathscr{A}^{(2)}$.

4.3. Results and analysis

Table 1 lists a brief classification, based on a scheme recommended by IEEE (IEEE 1044, 1994), of each fault and the number of test cases in the test case pool that detect it. Also listed is the number of pairwise $\mathscr{D}^{(1)}$'s which is the product of the number of failed tests and the number of successful tests. For example, of the 1000 test cases, 26 of them can detect F_{01}; this implies there are 974 successful tests. It also implies there are 25,324 (i.e., $26 * 974$) pairwise $\mathscr{D}^{(1)}$'s. Each pairwise $\mathscr{D}^{(1)}$ is examined to see whether it contains the corresponding fault. The percentage of bad $\mathscr{D}^{(1)}$'s is listed in the second column of Table 2, whereas the number of such $\mathscr{D}^{(1)}$'s for each fault can be computed as the product of this percentage and the total number of the pairwise $\mathscr{D}^{(1)}$'s from the last column of Table 1. The percentage of good $\mathscr{D}^{(1)}$'s can be obtained by subtracting the percentage of bad $\mathscr{D}^{(1)}$'s from 100; the number of good $\mathscr{D}^{(1)}$'s is the product of the good percentage and the total number of pairwise $\mathscr{D}^{(1)}$'s. Below, we first present the results and analysis for the bad $\mathscr{D}^{(1)}$'s followed by that for the good $\mathscr{D}^{(1)}$'s.

4.3.1. Augmenting with respect to bad $\mathscr{D}^{(1)}$'s

For each bad $\mathscr{D}^{(1)}$, the corresponding $\mathscr{A}^{(*)}$ is computed and examined to see whether it contains the corresponding fault. The percentage of good $\mathscr{A}^{(*)}$ is listed in the third column of Table 2. Let us use F_{01} as the example again. There are 312 (i.e., $1.23\% * 25{,}324$)[6] bad $\mathscr{D}^{(1)}$'s, but none of the

[6] Since only two decimal places are used, the product may be rounded up or down to an integer. Interested readers can contact the authors for the exact number of $\mathscr{D}^{(1)}$'s for each fault. The same applies to other computations that use the percentages expressed in this format.

Table 1
Classification, number of failed tests, and number of pairwise $\mathcal{D}^{(1)}$'s for each fault used in our study

	Fault classification		# of failed tests	# of pairwise $\mathcal{D}^{(1)}$'s
	Fault type	Subtype		
F_{01}	Logic omitted or incorrect	Missing condition test	26	25,324
F_{02}	Logic omitted or incorrect	Missing condition test	16	15,744
F_{03}	Computational problems	Equation insufficient or incorrect	36	34,704
F_{04}	Data handling problems	Data accessed or stored incorrectly	38	36,556
F_{05}	Logic omitted or incorrect	Forgotten cases or steps	35	33,775
F_{06}	Computational problems	Equation insufficient or incorrect	32	30,976
F_{07}	Computational problems	Equation insufficient or incorrect	32	30,976
F_{08}	Logic omitted or incorrect	Missing condition test	6	5964
F_{09}	Logic omitted or incorrect	Checking wrong variable	46	43,884
F_{10}	Logic omitted or incorrect	Checking wrong variable	320	217,600

Table 2
Percentage of bad $\mathcal{D}^{(1)}$'s and good $\mathcal{A}^{(*)}$'s, as well as the size distribution of $\mathcal{A}^{(*)}$'s

	% of bad $\mathcal{D}^{(1)}$'s	% of good $\mathcal{A}^{(*)}$	Size distribution of $\mathcal{A}^{(*)}$ in terms of % of the corresponding failed execution slice						
			[0,10]	(10,20]	(20,30]	(30,40]	(40,50]	(50,60]	(60,100]
F_{01}	1.23	100	1.60	11.21	39.10	46.47	1.60	0	0
F_{02}	0.61	100	30.20	25.00	33.33	11.45	0	0	0
F_{03}	4.98	100	0	0	0	0	83.62	16.37	0
F_{04}	78.69	100	0	1.47	14.40	31.28	12.30	39.24	1.27
F_{05}	52.12	92.76	0.53	14.02	35.05	17.06	18.57	14.74	0
F_{06}	0.62	100	0	31.77	55.20	13.02	0	0	0
F_{07}	0.62	100	0	31.77	55.20	13.02	0	0	0
F_{08}	2.82	100	1.19	24.40	51.78	22.61	0	0	0
F_{09}	2.62	99.39	3.06	18.02	44.88	31.32	2.71	0	0
F_{10}	22.06	86.82	0.53	13.87	41.46	31.33	9.26	3.51	0

$\mathcal{A}^{(*)}$'s are bad. The rest of Table 2 (columns four to ten) shows the size distribution of $\mathcal{A}^{(*)}$ in terms of the percentage of the corresponding failed execution slice. For example, there are 39.1% of all the $\mathcal{A}^{(*)}$'s for F_{01} with a size between 20% and 30% of the corresponding failed execution slice. From this table, we make the following observations:

- Three faults (F_{04}, F_{05} and F_{10}) have a significant number of bad $\mathcal{D}^{(1)}$'s, which implies that examining code only in these $\mathcal{D}^{(1)}$'s cannot locate the bugs. Additional code has to be included by using the method discussed in Section 2.1. The same applies to other faults even though they do not have as many bad $\mathcal{D}^{(1)}$'s as these three.
- The percentage of good $\mathcal{A}^{(*)}$ is either 100% or more than 99% except for F_{05} and F_{10} which have 92.76% and 86.82%, respectively. This implies that although the fault is not in $\mathcal{D}^{(1)}$, it will be, with a very high probability, in $\mathcal{A}^{(*)}$.
- The majority of $\mathcal{A}^{(*)}$ (such as those for F_{02}, F_{06}, F_{07}, F_{08} and F_{09}) have a size less than 30% of the corresponding failed execution slice. Others, such as those for F_{01}, F_{04}, F_{05} and F_{10}, have a size less than 40%. The only exception is F_{03} of which most $\mathcal{A}^{(*)}$ have a size between 40% and 50% of the corresponding failed execution slice.

Our data indicate that the augmentation method discussed in Section 2.1 is an effective approach to include additional code for locating a program bug when it is not in $\mathcal{D}^{(1)}$. Moreover, many of the resulting $\mathcal{A}^{(*)}$ only have a size between 10% and 20%, or 20% and 30%, or 30% and 40% of the corresponding failed execution slice.

4.3.2. *Refining with respect to good $\mathcal{D}^{(1)}$'s*

The size distribution of good $\mathcal{D}^{(1)}$'s in terms of percentage of the corresponding failed execution slice is listed in Fig. 4. The notation $F_k[\alpha, \beta]$ represents the set of all good $\mathcal{D}^{(1)}$'s for F_k with a size between α% and β% of the failed execution slice. From Fig. 4, we find that 31.45% of all the good $\mathcal{D}^{(1)}$'s for F_{01} have a size between 20% and 30% of the corresponding failed execution slice. Also, from Tables 1 and 2, there are $(100–1.23)\% * 25{,}324 = 25{,}012$ good $\mathcal{D}^{(1)}$'s. Together, there are $31.45\% * 25{,}012 = 7866$ $\mathcal{D}^{(1)}$'s in $F_{01}(20, 30]$. As explained in Footnote 6, since only two decimal places are used for representing these percentages, the exact number of $\mathcal{D}^{(1)}$'s may be rounded up or down to an integer. From this figure, we make the following observations:

- F_{04} has no good $\mathcal{D}^{(1)}$'s with a size less than 30% of the corresponding failed execution slice, that is, $F_{04}[0, 20] = 0$ and $F_{04}(20, 30] = 0$. Moreover, 96.12% of its good $\mathcal{D}^{(1)}$'s contain more than 40% of the corresponding failed execution slice.
- F_{05} and F_{10} have 40.67% and 31.8%, respectively, of all their good $\mathcal{D}^{(1)}$'s with a size greater than 40% of the corresponding failed execution slice.

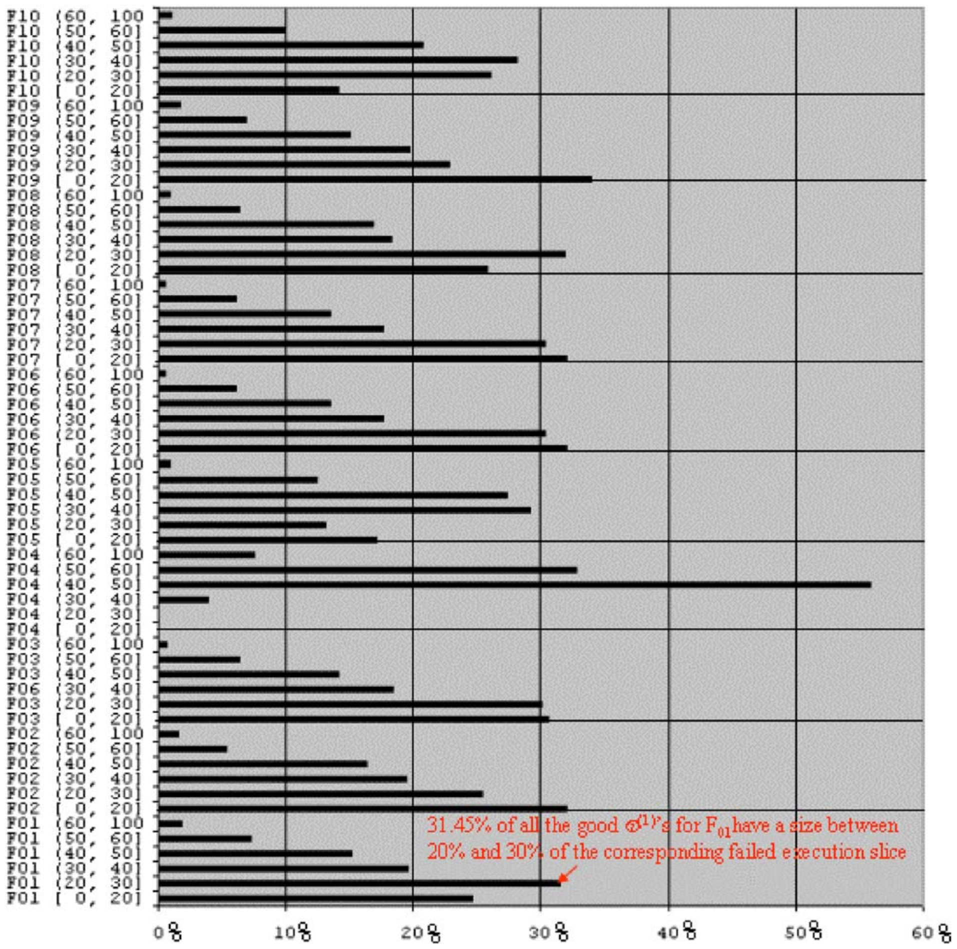


Fig. 4. Size distribution of good $\mathscr{D}^{(1)}$'s. For example, there are 31.45% of all the good $\mathscr{D}^{(1)}$'s for F_{01} with a size between 20% and 30% of the corresponding failed execution slice.

- Other faults have 20% (or a little more) of all the good $\mathscr{D}^{(1)}$'s with a size greater than 40% of the corresponding failed execution slice.

After a careful examination, we find that the above discrepancy is due to program structure and the location of the fault.

Given a good $\mathscr{D}^{(1)}$with respect to a specific fault (i.e., the fault is located in $\mathscr{D}^{(1)}$), if its size is less than 20% of the corresponding failed execution slice, we say this $\mathscr{D}^{(1)}$ is small enough[7] and can be used for effectively locating the fault. Otherwise, it should be further refined as it still contains too much code that needs to be examined. The refinement can be done by using additional successful tests as explained in Section 2.2.

Every good $\mathscr{D}^{(1)}$ in each category (such as $F_{01}(20,30]$, $F_{01}(30,40]$, etc.) is obtained by subtracting the execution slice of a successful test (say S_1) from the execution slice of a failed test (say F). Since there are too many such $\mathscr{D}^{(1)}$'s, we randomly select 20 of them from each category. For each selected $\mathscr{D}^{(1)}$, we do the following:

- Randomly select two more successful tests (say S_2 and S_3) such that S_1, S_2 and S_3 differ from each other.
- Construct the corresponding $\mathscr{D}^{(2)}$ by subtracting the execution slices of S_1 and S_2 from the execution slice of F, and $\mathscr{D}^{(3)}$ by subtracting the execution slices of S_1, S_2 and S_3 from the execution slice of F.
- Determine whether the corresponding fault is located in $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$.
- Compute the sizes of $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ in terms of percentage of the corresponding failed execution slice. Suppose $\mathscr{D}^{(1)}$, $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ have α%, β% and γ%. Compute also the differences $\alpha-\beta$ and $\alpha-\gamma$ which are, respectively, the size reductions due to the inclusion of the second and the third successful tests.

There should be 20 size reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(2)}$, and 20 between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(3)}$. Two averages are computed: one with respect to all the $\alpha-\beta$ and the other with respect to all the $\alpha-\gamma$.

We repeat the above procedures for good $\mathscr{D}^{(1)}$'s selected from each category. Since each $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ contains less

[7] If this is not the case (i.e., 20% of the failed execution slice still contains too much code), we can follow the same refining procedure explained in Section 2.2 to further prioritize code in this $\mathscr{D}^{(1)}$.

code than the corresponding $\mathscr{D}^{(1)}$, one question that we need to answer is whether such a size reduction has a negative impact on locating the fault. Fig. 5 shows the percentages of bad $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s with respect to each category which do not contain the fault. For example, 10% and 20% of the $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s in the category F_{08}(40, 50] do not contain F_{08}. Stated differently, of the 20 randomly selected good $\mathscr{D}^{(1)}$'s from this category, the refinement makes two of the resulting $\mathscr{D}^{(2)}$'s and four of the resulting $\mathscr{D}^{(3)}$'s bad because they do not contain the fault. From this figure, we observe that in many cases, the percentages of bad $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are 10% or lower. This implies most $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are good (i.e., they still contain the fault).

However, this is not true for F_{04} and F_{05} where the majority of $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are not good. To explain this, we use F_{04} as the example. In order to construct a good $\mathscr{D}^{(1)}$, $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ for this bug, we need to use one, two and three successful tests, respectively, which do not execute the code containing F_{04}. However, after a careful examination, we notice that most test executions will go through the code which contains F_{04}. As a result, not only the percentage of bad $\mathscr{D}^{(1)}$'s for F_{04} is high (78.69% as listed in Table 2), but the percentages of the corresponding bad $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are high as well. The same reason applies to F_{05} which has 52.12% of bad $\mathscr{D}^{(1)}$'s. As for F_{10}, we have mixed results. The percentages of bad $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are high in some cases but low for others. One reason for this is because 22.06% of its pairwise $\mathscr{D}^{(1)}$'s are bad. This implies that while constructing $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s, the chance to select a successful test which executes the code containing F_{10} is relatively high. In short, if the percentage of good $\mathscr{D}^{(1)}$'s is low, the chance for the corresponding $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s to be good is also low.

We now discuss the size reduction due to the refinement. Fig. 6 shows the average size reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(2)}$ (represented by the lengths of the black bars) and the reductions between $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ (represented by the lengths of the green bars or the white bars if the paper is printed in black and white). Since the green bars are appended at the right end of the black bars, the total length of the union of these two bars gives the reduction between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(3)}$. All these reductions are in terms of percentage of the corresponding failed execution slice. For example, as indicated in the figure, with respect to F_{08}(40, 50] the size reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(2)}$, and $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ are 18.45% and 9.54%, respectively. This also implies

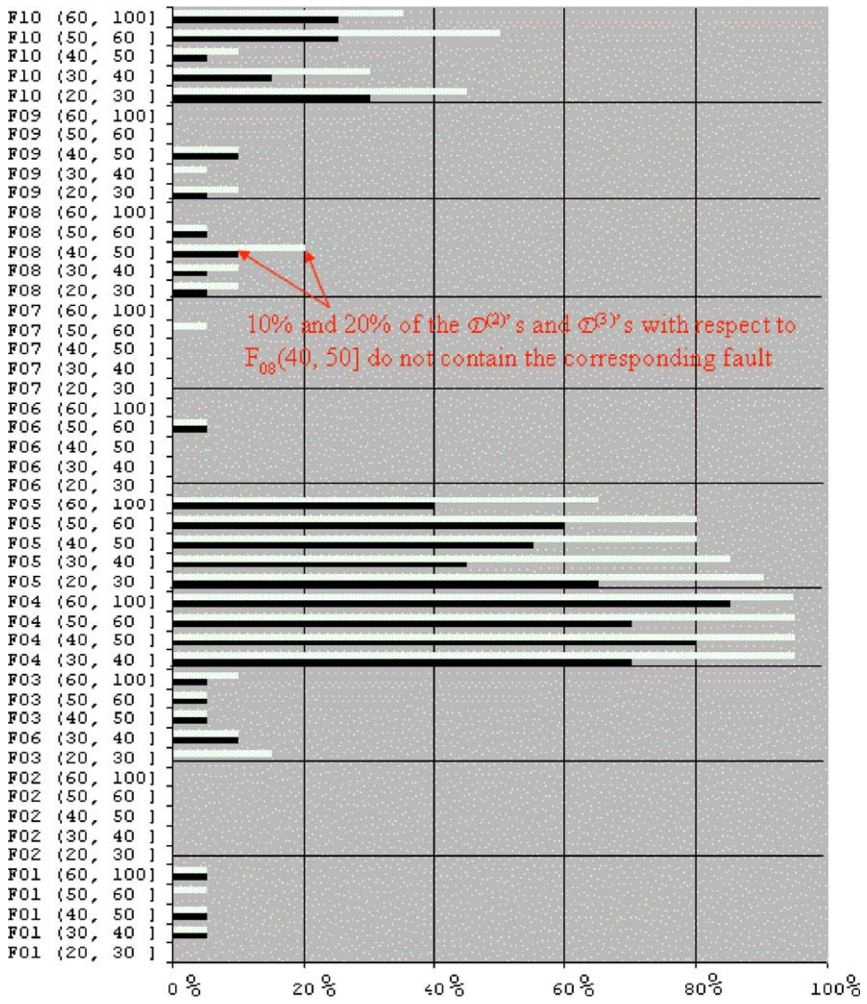


Fig. 5. Percentages of bad $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s which do not contain the corresponding fault.

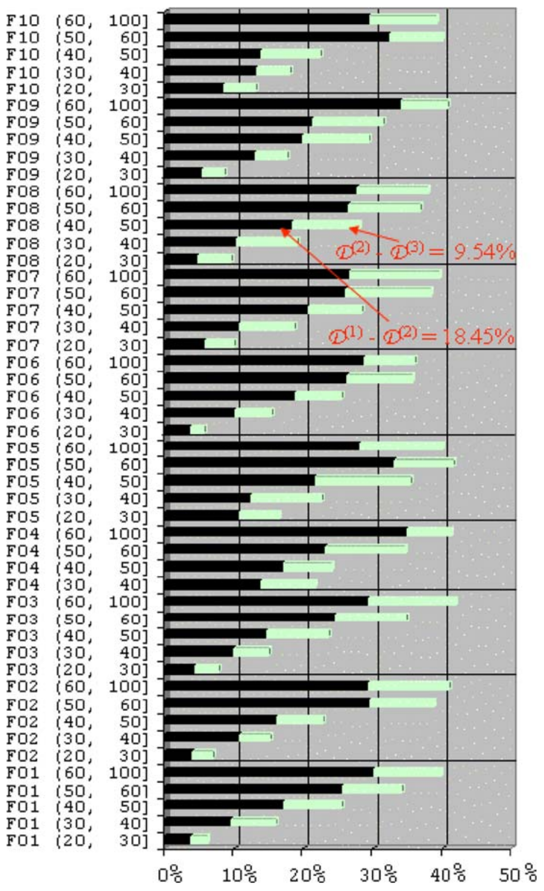


Fig. 6. Size reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(2)}$, and $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$. The reductions are in terms of percentage of the corresponding failed execution slice.

the reduction between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(3)}$ is 27.99%—the sum of 18.45% and 9.54% From this figure, we make the following observations:

- The size reductions of $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s are about the same for every fault.
- The reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(2)}$ are unanimously greater than those between $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ with no exception.
- In general, $\mathscr{D}^{(1)}$'s with more code have a larger size reduction than those with less code. For example, the size reductions between $\mathscr{D}^{(1)}$ and $\mathscr{D}^{(3)}$ in $\mathrm{F}_k(60, 100]$ and $\mathrm{F}_k(20, 30]$ are about 40% and 6–10%, respectively, with a few exceptions.

To conclude, our data indicate that in most cases $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s generated based on good $\mathscr{D}^{(1)}$'s are still good. That is, if $\mathscr{D}^{(1)}$ contains a program bug, then $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ are likely to contain the same bug as well. Some exceptions may occur, in particular when the percentage of good $\mathscr{D}^{(1)}$'s is low (i.e., too many pairwise $\mathscr{D}^{(1)}$'s are bad). Our data also indicate that the sizes of $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ can be much smaller than that of the corresponding $\mathscr{D}^{(1)}$ which implies that we can significantly reduce the code that need to be examined for locating a program bug. In short, the refining method discussed in Section 2.2 works for most good $\mathscr{D}^{(1)}$'s.

5. Discussion

We now discuss some important aspects related to the incremental debugging strategy proposed in Section 2.

First, the effectiveness of our debugging strategy depends on which successful tests are used. One good approach is to identify successful tests that are as similar as possible to the failed test (in terms of their execution slices) in order to filter out as much code as possible. In this way, we start the debugging with a very small set of suspicious code, and then increase the search domain, if necessary, based on the augmentation and refining methods described earlier. In addition, the effectiveness also depends on the nature of the bug and the structure of the program.

Second, if a program fails not because of the current test but because of another test which fails to set up an appropriate execution environment for the current test, then we should bundle these two test cases together as one single failed test. That is, the corresponding failed execution slice is the union of the execution slices of these two tests.

Third, for a bug introduced by missing code, our debugging strategy can still work because, for example, some unexpected code may be included in the failed execution slice. That is, such missing code has an adverse impact on a decision to cause a branch to be executed even though it should not be. If that is the case, the abnormal execution path with respect to a given test can certainly help us realize that some code required for making the right decision is missing.

Finally, since testing and debugging are very closely related, a common practice is to fix the underlying program and remove the corresponding bug(s) as soon as some abnormal behavior is detected during the testing. Under this situation, programmers have one failed test in conjunction with some successful tests to be used for prioritizing code from the debugging point of view. This is the scenario discussed in this paper. However, a programmer may decide not to fix the bug right away for certain specific reasons. If so, (s)he may observe additional failures caused by the same fault(s) in the subsequent executions. That is, for the given fault(s), there exists more than one failed test. If that is the case, instead of defining E_{F} as the execution slice of a single failed test (refer to Section 2), the programmer may define it as the intersection of the execution slices of multiple failed tests. For example, E_{F} can be defined as the intersection of the execution slice of a failed test t_α and that of another failed test, t_β. This is because the code that is executed by both t_α and t_β is more likely to contain the bug than the code which is only executed by t_α or t_β but not both (Wong et al., 2005).

6. Related studies

Program slicing (Weiser, 1984) is a commonly used technique for debugging (Weiser, 1982). Since a static slice might contain too much code to be debugged, Lyle and Weiser propose constructing a program dice (as the set difference of two groups of static slices) to further reduce the search domain for possible locations of a fault (Lyle and Weiser, 1987). An alternative is to use dynamic slicing (Agrawal and Horgan, 1990; Korel and Laski, 1988) instead of static slicing, as the former can identify the statements that indeed do, rather than just possibly could (as by the latter), affect a particular value observed at a particular location. A third alternative is to use execution slicing and dicing to locate faults in C programs (Agrawal et al., 1995) or software architecture design in SDL (Wong et al., 2005). The reason for choosing execution slicing in this paper over static slicing or dynamic slicing is that a static slice focuses on finding code that could possibly have an impact on the variables of interest for *any* inputs instead of code that indeed affects those variables for a *specific* input. This implies a static slice does not make any use of the input values that reveal the fault. It clearly violates a very important concept in debugging which suggests that programmers analyze the program behavior under the test case that reveals the fault, not under a generic test case. The disadvantage of using dynamic slices is that collecting them may consume excessive time and file space, even though different algorithms have been proposed to address these issues. On the other hand, for a given test case, its execution slice is the set of code (basic blocks in our case) executed by this test. Such a slice can be constructed very easily. This is particularly true if we know the code coverage of a test because the corresponding execution slice of this test can be obtained simply by converting the coverage data collected during the testing into another format, i.e., instead of reporting the coverage percentage, it reports which basic blocks are covered.

Another debugging technique is to use execution backtracking. The major problem with this approach is that it may require a large amount of space to save the execution history and program states. To solve this problem, Agrawal et al. propose a structured backtracking approach without storing the entire execution history (Agrawal et al., 1993). They present a debugging technique based on dynamic slicing and backtracking, where a dynamic slice contains all the statements that actually affect the value of a variable at a program point for a particular execution of the program, and backtracking is used to restore the program state at a given location without having to re-execute the program from the beginning. More specifically, after a program failure is detected, its external symptoms are translated into the corresponding internal symptoms in terms of data or control problems in the program. Then one of the internal symptoms is selected for creating a dynamic slice. The next step is to guess the possible location of the fault that is responsible for the failure. This can be done by selecting a statement in the slice, restoring the program state to the state when the control last reached that statement, and examining values of some variables in the restored state. If this does not reveal the fault, one has three options: (1) further examining the restored state, (2) making a new guess by selecting another statement in the slice and restoring the program state accordingly, or (3) creating a new dynamic slice based on a different internal symptom. These steps are repeated until the fault is localized. A prototype tool, SPYDER, was implemented to support this technique. One problem with this approach is that translating externally visible symptoms of a failure into corresponding internal program symptoms can be very difficult in some cases. Another problem is that slices do not make explicit dependences among multiple occurrences of the same statement. The third problem is that execution backtracking has its limitations, in particular, if the execution of a statement can have impacts outside the boundaries of the program, e.g., into the operating system. Under this condition, only a partial backtracking is possible.

Korel presents a prototype of a fault localization assistant system (PELAS) to help programmers debug Pascal programs (Korel, 1988). He claims that PELAS is different from others in that it makes use of the knowledge of program structure to guide a programmer's attention to possible erroneous parts of the program. More specifically, such knowledge is derived from the program and its execution trace and is represented by a dependence network. This network is then used to discover probable sources of the fault(s). The rationale of the entire approach is that any program instructions in the execution trace from the beginning to a position of incorrectness could conceivably be responsible for the abnormal behavior of the program being debugged. However, his approach does not include the *refining* and *augmentation* methods as described in this paper.

Jones et al. describe a visualization technique to assist in fault location by illuminating the statements of a given program in different colors (Jones et al., 2002). For each statement, its color depends on the relative percentage of the successful tests that execute the statement to the failed tests that execute the statement. If the statement is executed by more successful tests than failed tests, its color appears to be greener, whereas the other way around (i.e., more failed tests than successful tests) makes the color redder. Two measurements are computed: a false negative in terms of how often a faulty statement is not colored in red, and a false positive in terms of how often a non-faulty statement is colored in red. Their results show that for some faults, this technique works well. However, for other faults, especially those in the initialization code or on the main path in the program which are executed by most of the test cases, none of the faulty statements are colored in the red range. A possible way to overcome this problem is to use slice and data dependency analysis.

7. Conclusion and future research

We propose a debugging method based on execution slices and inter-block data dependency for effectively locating program bugs. We start with code that is only executed by the failed test but not any successful tests. To do so, we first construct $\mathscr{D}^{(1)}$ which is obtained by subtracting the execution slice of a successful test from the execution slice of a failed test. Then, we follow the *refining* procedure discussed in Section 2.2 to construct, for example, $\mathscr{D}^{(2)}$ and $\mathscr{D}^{(3)}$ by subtracting the union of the execution slices of two and three successful tests, respectively, from the execution slice of a failed test. Since the likelihood of a piece of code containing a bug decreases if it is executed by some successful tests, we will try to locate a bug by first examining $\mathscr{D}^{(3)}$ followed by $\mathscr{D}^{(2)}$ and then $\mathscr{D}^{(1)}$. If we cannot find the bug in $\mathscr{D}^{(1)}$, we follow the *augmenting* procedure discussed in Section 2.1 to first construct $\mathscr{A}^{(1)}$ which is directly data dependent on $\mathscr{D}^{(1)}$. In other words, $\mathscr{A}^{(1)}$ contains code that either defines variables used in $\mathscr{D}^{(1)}$ or uses variables defined in $\mathscr{D}^{(1)}$. We examine code in $\mathscr{A}^{(1)}$ to see whether it contains the bug. If not, we will construct $\mathscr{A}^{(2)}$ which is the union of $\mathscr{A}^{(1)}$ and the set of code that is directly data dependent on $\mathscr{A}^{(1)}$. If the bug is in $\mathscr{A}^{(2)}$, we have successfully located it. Otherwise, we continue this process until $\mathscr{A}^{(*)}$ is constructed and examined. If the bug is still not found, we then examine the last piece of code, namely, code in the failed execution slice but not in $\mathscr{A}^{(*)}$ and $\mathscr{D}^{(1)}$. In short, we propose an incremental debugging strategy to first examine the most suspicious code identified by using some heuristics supported by good reasoning. If the bug is not located then we increase the search domain step by step by including additional code that is next most likely to contain the bug.

We conducted a case study on a SPACE application to illustrate the effectiveness of our method in locating program bugs. A tool, DES*i*D, was implemented to automate the slice generation, code prioritization, and data collection. Our results are encouraging. They indicate that if a program bug is in $\mathscr{D}^{(1)}$, then the chance for $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s generated based on this $\mathscr{D}^{(1)}$ to contain the same bug is also high in most cases with some exceptions as discussed in Section 4. In addition, the sizes of $\mathscr{D}^{(2)}$'s and $\mathscr{D}^{(3)}$'s can be much smaller than that of the corresponding $\mathscr{D}^{(1)}$. On the other hand, the method we used for augmenting a bad $\mathscr{D}^{(1)}$ (i.e., a $\mathscr{D}^{(1)}$ that does not contain the bug) is also effective for locating a program bug because the chance for the corresponding $\mathscr{A}^{(*)}$ to contain the bug is very high and yet its size is relatively small in comparison to the failed execution slice.

Finally, since execution slices obtained as well as $\mathscr{D}^{(1)}, \mathscr{D}^{(2)}, \mathscr{D}^{(3)}, \ldots, \mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots$, and $\mathscr{A}^{(*)}$ depend on the test cases used, different suspicious code will be prioritized differently by different sets of successful and failed tests. An interesting future study would be to develop guidelines (in addition to what has been discussed in Section 5) to help programmers select appropriate successful test(s) with respect to a given failed test so that only a small portion of the program needs to be examined for locating a bug. An intuitive guess is that such selection will depend on the nature of the bug to be located as well as the structure of the program to be debugged. Another important future study is to apply our debugging method to industry projects to examine how much time programmers can save by using our method in locating bugs in comparison to other methods.

References

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, New York, USA, June, pp. 246–256.

Agrawal, H., DeMillo, R.A., Spafford, E.H., 1993. Debugging with dynamic slicing and backtracking. Software—Practice and Experience 23 (6), 589–616.

Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, Toulouse, France, October, pp. 143–151.

Cancellieri, A., Giorgi, A., 1994. Array Preprocessor User Manual. Technical Report IDS-RT94/052.

Horgan, J.R., London, S.A., 1991. Data flow coverage and the C language. In: Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification, Victoria, British Columbia, Canada, October, pp. 87–97.

IEEE., 1044—standard classification for software errors, faults and failures, 1994. IEEE Computer Society.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, Florida, USA, May, pp. 467–477.

Korel, B., 1988. PELAS—program error-locating assistant system. IEEE Transactions on Software Engineering 14 (9), 1253–1260.

Korel, B., Laski, J., 1988. Dynamic program slicing. Information Processing Letters 29 (3), 155–163.

Lyle, J.R., Weiser, M., 1987. Automatic program bug location by program slicing. In: Proceedings of the 2nd International Conference on Computer and Applications, Beijing, China, June, pp. 877–883.

Musa, J.D., 1993. Operational profiles in software reliability engineering. IEEE Software 10 (2), 14–32.

χSuds User's Manual, 1998. Telcordia Technologies.

Weiser, M., 1982. Programmers use slices when debugging. Communications of the ACM 25 (7), 446–452.

Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering 10 (4), 352–357.

Wong, W.E., Sugeta, T., Qi, Y., Maldonado, J.C., 2005. Smart debugging software architectural design in SDL. Journal of Systems and Software 76 (1), 15–28.

W. Eric Wong received his B.S. in Computer Science from Eastern Michigan University, and his M.S. and Ph.D. in Computer Science from Purdue University. In 1997, he received the Quality Assurance Special Achievement Award from Johnson Space Center, NASA. Currently, he is a tenured associate professor in Computer Science at the University of Texas at Dallas. Before joining UTD in 2002, he was with Telcordia Technologies (formerly Bellcore) as a senior research scientist and a project manager in charge of the initiative for Dependable Telecom Software Development. His research focus is on the development of techniques to help practitioners produce high quality software at low cost. In particular, he is doing research in the areas of software testing, reliability, metrics, and QoS at the application as well as architectural design levels. He served as general chair for ICCCN 2003 and program chair for Mutation 2000, ICCCN 2002, COMPSAC 2004, SEKE 2005, and ISSRE 2005. He is a senior member of IEEE.

Yu Qi received his B.S. in Computer Science from Wuhan University in China and M.S. in Computer Science from Fudan University in China. Currently, he is a Ph.D. candidate in Computer Science at the University of Texas at Dallas. His research interests are in software testing and embedded systems.