FLAVS: A Fault Localization Add-in for Visual Studio

Nan Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
wangnangg@gmail.com

Zheng Zheng
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
Zhengz2011@gmail.com

Zhenyu Zhang*
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

Cheng Chen
School of Computer Science and Engineering
Beihang University
Beijing, China
651044554@qq.com

Abstract—**Dynamic fault localization is a representative concept and product proposed by academia to alleviate software engineering pains, but it is rarely heard adopted or used in realistic development. Realizing the difficulties in transferring the approaches of dynamic fault localization to practical tools, this paper gives our work FLAVS, whose add-in implementation organically and seamlessly integrates the approach of dynamic fault localization with software IDEs. The tool is useful for developers using Microsoft Visual Studio platform to debug and test programs with complex bugs. Besides, it is also valuable for researchers to design new fault localization methods and draw performance comparison among different method candidates.**

Keywords- fault localization; Microsoft Visual Studio add-in; fault localization tools

I. INTRODUCTION

Software applications appear in every corner of our daily lives. However, software application is still far from bug-free and the resultant software failures are continuously affecting the quality of software in use. Program debugging is the process of locating faults in faulty programs, repairing the faults located, and re-testing the repaired programs [13]. It is often a lengthy and manual procedure in practice, and there always exist attempts to look for automatic mechanisms to facilitate this task. Automatic program fault localization refers to the mechanism to locate faults in programs. Typical techniques include Tarantula [7], CBI [8], SOBER [10], Delta Debugging [15], Predicate Switching [16], CP [18], and so on (e.g., [3][16]).

Among the many approaches to automatically locate faults in faulty programs, coverage-based fault localization techniques (CBFL) form a big family. The basic intuition behind is that the exercising of fault-relevant program units is correlated with the occurrence of program failures. CBFL techniques estimate the suspiciousness of program units by contrasting their coverage information collected in successful executions and those collected in failed executions, and narrow down the fault search region by focusing on the units having great differences. The program debugging difficulties can be alleviated if programmers follow the fault localization suggestions to check the program for faults.

The spreading and applying of coverage-based fault localization techniques in industrial activities have at least four obstacles. First, to drive such a technique, oracle and dynamic program spectra are two necessary conditions, which are not directly available in most realistic cases. Second, to support locating complex bugs like Mandelbugs [4], workload designing, environmental factor monitoring and system long-term running are necessary, which are not implemented in almost all existent fault localization tools. Third, after locating a fault and fixing it, there is a need to rerun all the test cases to ensure the quality of fixing. Such a step is mostly finished by error-prone human work. Fourth, the fault localization techniques are developing rapidly for decades of years. Any attempt to stick to an ideal algorithm cannot be pervasively effective.

To automate the fault localization process, we developed an add-in for the most popular development platform Microsoft Visual studio. Once the program under test is complied, FLAVS instruments the source code in advance. When the program under test executes, FLAVS records arguments, parameters, and standard inputs for replaying. During the program's execution, FLAVS continuously logs the execution information including statement coverage, call stack traces, environmental factors (For example, memory consuming, CPU usage, and thread numbers) and so on. After a program run finishes, the developer is asked to mark the status of the program run, i.e., successful or failed. FLAVS then employs a fault localization module to incrementally update the suspiciousness of each program statement being related to fault, lists out the executed statements in the FLAVS window, and highlights the most suspicious ones in appropriate colors. FLAVS also allows developer to jump to the right position in the source code window when clicking on a suspicious statement in the FLAVS window. After locating and fixing a fault, FLAVS

* Corresponding author: Zhenyu Zhang (mail: zhangzy@ios.ac.cn).

provides a short cut to automatically rerun all the previously recorded test cases to conduct regression testing.

The main features of FLAVS are as follows. (1) It provides program fault localization assistance by visualizing suspicious program units. (2) It supports workload designing and system long-term execution. (3) It can monitor environmental factors. (4) It automates the regression testing process in a record and replay manner. (5) It allows new algorithm designing. (6) It is integrated into a widely used IDE, i.e. Visual Studio.

This paper presents the second release of our tool. The rest of the paper is organized as follows. Section II gives related work. Its functionalities and implementation are presented in Section III. Section IV demonstrates two typical usage scenarios. Section V concludes the paper and presents the future work.

II. Related work

Previous CBFL studies propose different models to assess the similarities between program coverage for different executions, to determine the suspicious program units. Typically, such a similarity coefficient calculates a suspiciousness score of a program unit using four parameters, namely, the number of passed executions that exercise the program unit (a_{ep}), the number of failed executions that exercise it (a_{ef}), the number of passed executions that do not exercise it (a_{np}), and the number of failed executions that do not exercise it (a_{nf}). For example, the classical CBFL technique Tarantula [7] uses the formula $\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$ to estimate the suspiciousness of a program unit.

CBFL techniques assess the suspiciousness of program units and generate a ranked list of all suspicious units to help locating faults. Possible program units include statements, functions, paths, data-dependency pairs and so on. Harrold et al. [4] evaluated nine kinds of program units, among which, the execution trace is the most widely used one. Recently, a trend is to focus on the coefficient itself to simplify the fault localization problem [12] [16]. For example, Naish et al. [12] listed many CBFL techniques and compared them using the same program settings. In our tool, we implemented all of them and provided an algorithm designing mechanism for FLAVS to produce new fault localization modules by users. Compared to these works, our work focus on the efforts of converting the approaches of dynamic fault localization into a practical tool embedded in a popular IDE.

There exist a few toolsets or prototypes for fault localization, such as Crisp[2], Zoltar [6], Falcon [13], χDebug [17], DES*i*D [18], and also some fault localization tools mainly concerning special applications, such as FaultySheet Detective [1]. Our tool is different from them basically in that FLAVS integrates fault localization with a popular development platform, enabling its easy deployment in the code development phase. FLAVS also provides regression testing supports, and can be used in developing new fault localization algorithms.

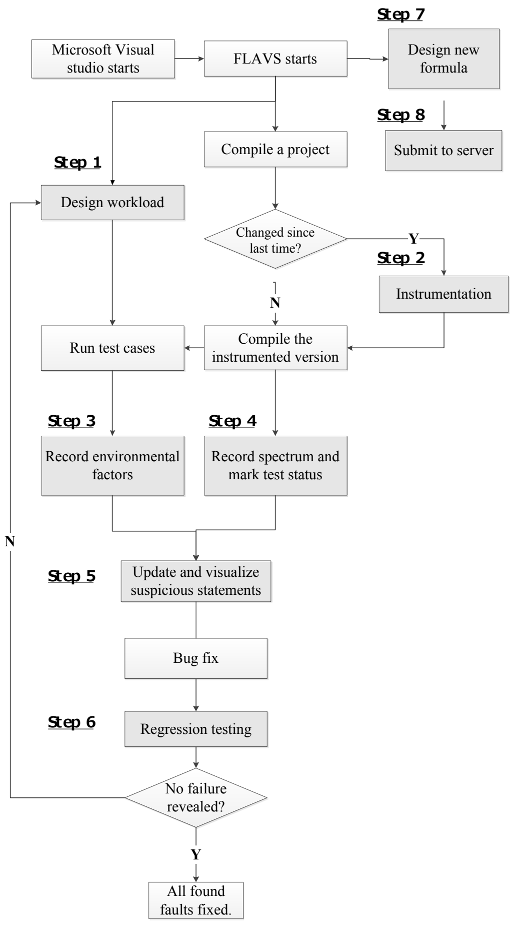


Fig.1 Flowchart of FLAVS

To the best of our knowledge, there are only a few add-in fault localization tools [5] [8]. They were all implemented in Eclipse IDE. Comparing with these works, another important feature of FLAVS is its consideration of Complex bugs like Mandelbugs [20]. Much of the efforts in software fault localization have been taken on locating and removing software bugs known as Bohrbugs as opposed to more-difficult-to-reproduce Mandelbugs. Comparing with Mandelbugs, Bohrbugs are bugs in software that are easy to reproduce and debug; they do not change behavior as the system state changes. Mandelbug activation, on the other hand, depends not only on the input of the program but also on different environmental factors, such as the state of operating system resources, concurrency with other processes, hardware and software interactions, and other factors. To facilitate the debugging of Mandelbugs, the functions, such as workload designing, environmental factor monitoring and long-term running are provided in our tools.

III. FLAVS Overview

A. Flowcharts and Functions

In this section, we will present the main functions of FLAVS. Fig. 1 shows the flowchart of FLAVS, which include the relations among the functions. The illustration of the functions can be found in Section IV for detail.

Function 1: Fault localization assistance

[Step 1] Workload Design

Workload is the amount of test cases that a test has to do. In this step, the set of test cases executed, the times of executing each test case, and the running intervals among test cases can be designed by users.

[Step 2] Code instrumentation

To collect dynamic spectrum information from program execution, instrumentation is needed. FLAVS provides an automatic instrumentation mechanism. It is triggered before the compiling of the program.

[Step 3] Environmental factor gathering

To handle complex bugs like Mandelbugs, FLAVS monitors the environmental factors of the running project. The factors include memory consuming, CPU usage, thread numbers and so on.

[Step 4] Spectrum and mark test status recording

FLAVS provides a unified entry to set arguments, and parameters to start debugging. It can record the coverage information automatically. Once the program exits, it can automatically or manually mark the test status (successful or failed).

[Step 5] Fault location calculation and visualization

From the automatically collected spectrum, FLAVS calculates the suspiciousness of each statement, predicate or function. The suspicious units are shown and highlighted in the FLAVS window and users can jump to the right position in the source code window by clicking on them.

Function 2: Regression testing

[Step 6] Rerun all the captured test cases

When a fault is located and fixed, FLAVS provides a short cut to rerun all the previous recorded test cases to check the quality of fixing. The execution results of rerun are shown in a list. The suspicious units in the FLAVS window are updated afterwards.

Function 3: Designing new algorithm

[Step 7] Design a new technique

The users can design their own techniques from scratch or from a template of existing techniques. Since the entire fault localization infrastructure has already been setup by our tool, the users only need to implement a core formula. Using the provided interface, the users can compose any innovative formulas, and verify their effectiveness.

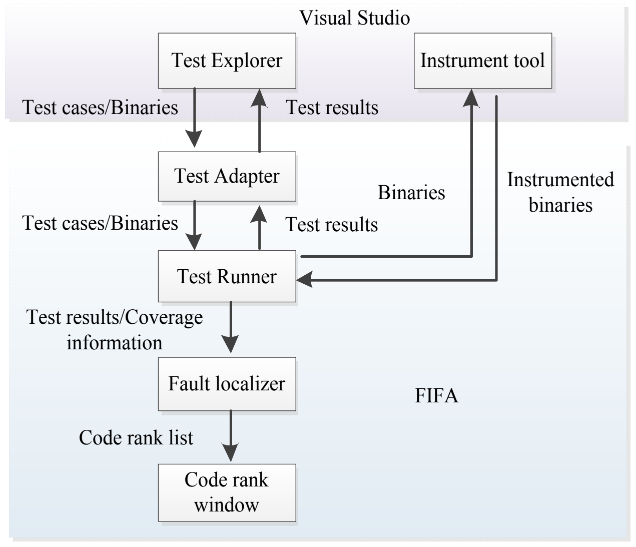


Fig. 2. Framework of the VS Add-in

[Step 8] Submit to the server

A stable version of any user-designed new formula can be submitted to servers. A user access mechanism is scheduled in the future work.

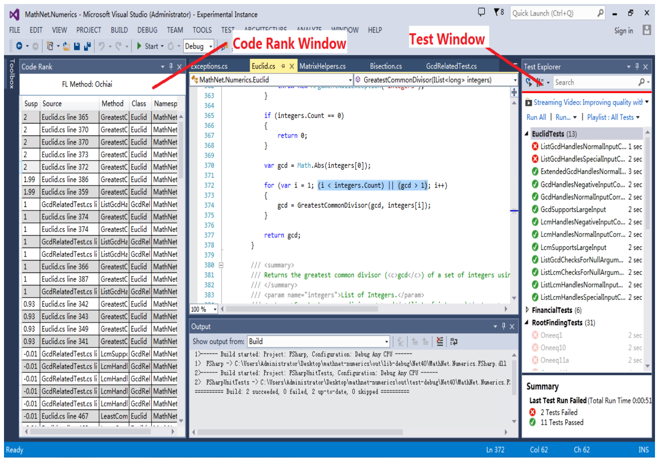
B. FLAVS Implementation

FLAVS is a Visual Studio add-in coded in C#. Microsoft Visual Studio is one of the most popular IDEs, which can be used to develop projects in Visual C++, Visual Basic, C#, and other languages. It provides add-in APIs and makes the retrieval of program testing and debugging information available.

Figure 2 shows the framework of the add-in and its interactions with Visual Studio. The add-in communicates with the Visual Studio unit testing engine to capture the test cases and binaries, and returns test results to Visual Studio. To facilitate the communication, a test adapter is developed. Besides, the add-in interacts with an instrument tool provided by Visual Studio to generate the instrumented binaries of the test object. After obtaining the test cases and instrumented binaries, the test runner is triggered. As a result, the test results and the coverage information are generated. Afterwards, FLAVS makes use of the specified fault localization technique to calculate the suspiciousness of each program unit and shows the resulting rank list in the code rank window. A user then refers to the visualized fault localization result to find the fault.

In FLAVS, a fault localization formula file is coded in XML and provided at the server side. A XML parser is invoked by FLAVS to update formulas. Test cases are recorded as inputs and expected outputs. The former consists of command line arguments and standard input contents. In the future work, program inputs in the form of keyboard and mouse events will be supported. The latter is supposed to be specified by the programmer in plain text format. It will be compared with the standard output to serve as oracles. FLAVS communicates with Visual Studio Debugging routines to capture the signal of program exit, so that when no expected output is given, the programmer will be asked to manually mark the test status, i.e., either successful or failed, when program exits.

New algorithm can be designed by using specific template, where six parameters, P, F, a_{ef}, a_{ep}, a_{nf}, and a_{np} are manipulated (any operator can be used to connect them) to form a meaningful formula. The notation of the parameters can be found in [16]. A parser is invoked to justify the syntactic correctness of the algorithm.


MathNet.Numerics - Microsoft Visual Studio (Administrator) - Experimental Instance
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start
Debug
Quick Launch (Ctrl+Q)
Sign in
Code Rank Window
Test Window
Toolbox
Code Rank
FL Method: Ochiai
Exceptions.cs
Euclid.cs
MatrixHelpers.cs
Bisection.cs
GcdRelatedTest.cs
MathNet.Numerics.Euclid
GreatestCommonDivisor(IList<long> integers)
100 %
Test Explorer
Search
Streaming Video: Improving quality with
Run All | Run... | Playlist : All Tests
Ready
Ln 372
Col 62
Ch 62
INS

Susp	Source	Method	Class	Namesp
2	Euclid.cs line 365	GreatestC	Euclid	MathNet
2	Euclid.cs line 370	GreatestC	Euclid	MathNet
2	Euclid.cs line 370	GreatestC	Euclid	MathNet
2	Euclid.cs line 373	GreatestC	Euclid	MathNet
2	Euclid.cs line 372	GreatestC	Euclid	MathNet
1.99	Euclid.cs line 386	GreatestC	Euclid	MathNet
1.99	Euclid.cs line 359	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	Euclid.cs line 374	GreatestC	Euclid	MathNet
1	Euclid.cs line 374	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	Euclid.cs line 366	GreatestC	Euclid	MathNet
1	Euclid.cs line 387	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
0.93	Euclid.cs line 342	GreatestC	Euclid	MathNet
0.93	Euclid.cs line 343	GreatestC	Euclid	MathNet
0.93	Euclid.cs line 349	GreatestC	Euclid	MathNet
0.93	Euclid.cs line 341	GreatestC	Euclid	MathNet
-0.01	GcdRelatedTest.cs li	LcmSuppo	GcdRel	MathNet
-0.01	GcdRelatedTest.cs li	LcmHandl	GcdRel	MathNet
-0.01	GcdRelatedTest.cs li	LcmHandl	GcdRel	MathNet
-0.01	GcdRelatedTest.cs li	LcmHandl	GcdRel	MathNet
-0.01	GcdRelatedTest.cs li	LcmHandl	GcdRel	MathNet
-0.01	Euclid.cs line 467	LeastCom	Euclid	MathNet

```
363             }
364 
365             if (integers.Count == 0)
366             {
367                 return 0;
368             }
369 
370             var gcd = Math.Abs(integers[0]);
371 
372             for (var i = 1; (i < integers.Count) || (gcd > 1); i++)
373             {
374                 gcd = GreatestCommonDivisor(gcd, integers[i]);
375             }
376 
377             return gcd;
378         }
379 
380         /// <summary>
381         /// Returns the greatest common divisor (<c>gcd</c>) of a set of integers usin
382         /// </summary>
383         /// <param name="integers">List of Integers.</param>
```

Output

Show output from: Build

```
1>------ Build started: Project: FSharp, Configuration: Debug Any CPU ------
1>  FSharp -> C:\Users\Administrator\Desktop\mathnet-numerics\out\lib-debug\Net40\MathNet.Numerics.FSharp.dll
2>------ Build started: Project: FSharpUnitTests, Configuration: Debug Any CPU ------
2>  FSharpUnitTests -> C:\Users\Administrator\Desktop\mathnet-numerics\out\test-debug\Net40\MathNet.Numerics.F
========== Build: 2 succeeded, 0 failed, 2 up-to-date, 0 skipped ==========
```

EuclidTests (13)

✗	ListGcdHandlesNormalInputC...	1 sec
✗	ListGcdHandlesSpecialInputC...	2 sec
✓	ExtendedGcdHandlesNormalI...	3 sec
✓	GcdHandlesNegativeInputCo...	2 sec
✓	GcdHandlesNormalInputCorr...	2 sec
✓	GcdSupportsLargeInput	2 sec
✓	LcmHandlesNegativeInputCo...	2 sec
✓	LcmHandlesNormalInputCorr...	2 sec
✓	LcmSupportsLargeInput	2 sec
✓	ListGcdChecksForNullArgum...	2 sec
✓	ListLcmChecksForNullArgum...	2 sec
✓	ListLcmHandlesNormalInput...	2 sec
✓	ListLcmHandlesSpecialInputC...	2 sec

FinancialTests (6)

RootFindingTests (31)

✗	Oneeq1	2 sec
✗	Oneeq10	2 sec
✗	Oneeq11a	2 sec

Summary

Last Test Run Failed (Total Run Time 0:00:51

✗ 2 Tests Failed

✓ 11 Tests Passed

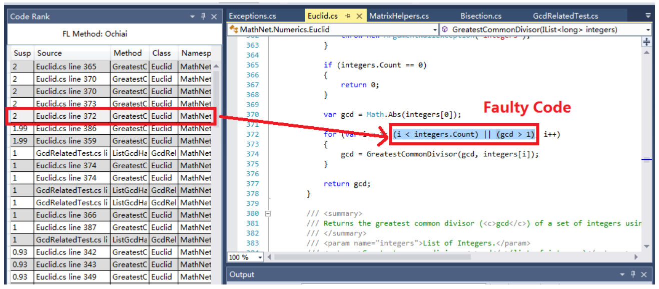
Fig. 3. GUI of FLAVS

IV. EXAMPLES

In this section, two examples will be used to illustrate the usage of FLAVS. One is a widely used open-source program, Math.NET Numerics, and the other is a program with an injected dead-lock bug.

A. *Example 1: Math.NET Numerics*

Once launching Microsoft Visual Studio, the add-in of FLAVS is loaded automatically. The user can at any time enable the code rank window of the add-in from the menu. Center of Fig. 3 is the code review window, which shows the program Math.NET Numerics. The program aims to provide methods and algorithms for numerical computations in science, engineering and everyday use. It is programmed



Susp	Source	Method	Class	Namesp
2	Euclid.cs line 365	GreatestC	Euclid	MathNet
2	Euclid.cs line 370	GreatestC	Euclid	MathNet
2	Euclid.cs line 370	GreatestC	Euclid	MathNet
2	Euclid.cs line 373	GreatestC	Euclid	MathNet
2	Euclid.cs line 372	GreatestC	Euclid	MathNet
1.99	Euclid.cs line 386	GreatestC	Euclid	MathNet
1.99	Euclid.cs line 359	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	Euclid.cs line 374	GreatestC	Euclid	MathNet
1	Euclid.cs line 374	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
1	Euclid.cs line 366	GreatestC	Euclid	MathNet
1	Euclid.cs line 387	GreatestC	Euclid	MathNet
1	GcdRelatedTest.cs li	ListGcdHa	GcdRel	MathNet
0.93	Euclid.cs line 342	GreatestC	Euclid	MathNet
0.93	Euclid.cs line 343	GreatestC	Euclid	MathNet
0.93	Euclid.cs line 349	GreatestC	Euclid	MathNet

```
363             }
364 
365             if (integers.Count == 0)
366             {
367                 return 0;
368             }
369 
370             var gcd = Math.Abs(integers[0]);
371 
372             for (var i = 1; (i < integers.Count) || (gcd > 1); i++)
373             {
374                 gcd = GreatestCommonDivisor(gcd, integers[i]);
375             }
376 
377             return gcd;
378         }
379 
380         /// <summary>
381         /// Returns the greatest common divisor (<c>gcd</c>) of a set of integers usin
382         /// </summary>
383         /// <param name="integers">List of Integers.</param>
```

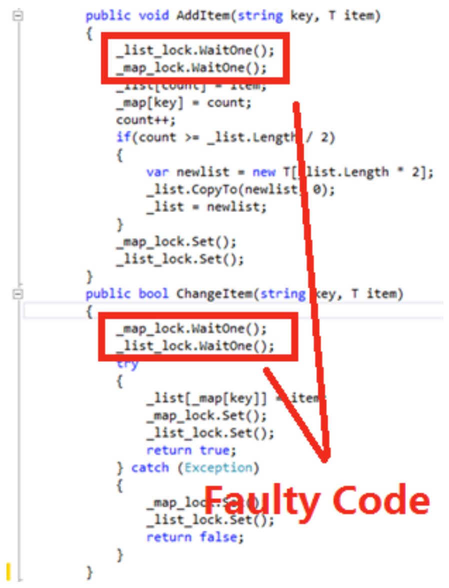
Fig. 4. Code rank window of FLAVS

with C#, consisting of more than 500 files and 120 thousands of executable statements. The program can be downloaded from [11].

Though fault localizations and regression testing are conducted in background, a user can show the test window and code rank window at any time. The program outputs are retrieved and compared with the expected outputs. For some of them the observed outputs are consistent with the expected outputs, FLAVS determines that they are successful runs and marks “Yes” in the corresponding cells. Otherwise, a failed run is found and FLAVS marks “No” for it. Note that when no expected output is given, the result is left to the programmer to manually decide, or left as “Unknown” by default.

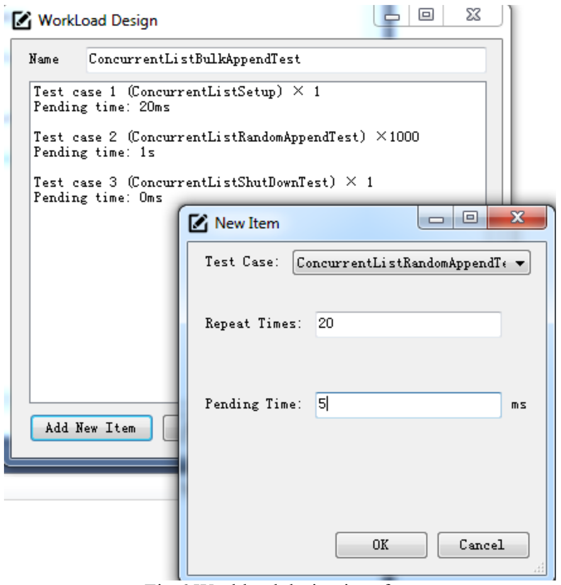
After identifying test results and obtaining the coverage information, FLAVS starts a fault localization process to calculate the suspiciousness for each statement. The fault localizing results are automatically updated and showed in the code rank window (See Fig. 4). The basic blocks are ranked in descending order. In this case, the user checks each of the highly suspicious statements, and finds out that the 4th highest suspicious one, which is the 372nd statement, is the root cause of the observed failures. The fault is actually a logic error, specifically “&&” is written to “||” by mistake. The user can jump to the faulty position by clicking the item in the rank list. After fixing the fault and compiling the program, the tester can perform a regression test. Note that, in this example we use all the test cases downloaded from [11].

B. *Example 2: A program injected with a Dead Lock bug*



```
public void AddItem(string key, T item)
{
    _list_lock.WaitOne();
    _map_lock.WaitOne();
    _list[count] = item;
    _map[key] = count;
    count++;
    if(count >= _list.Length / 2)
    {
        var newlist = new T[_list.Length * 2];
        _list.CopyTo(newlist, 0);
        _list = newlist;
    }
    _map_lock.Set();
    _list_lock.Set();
}
public bool ChangeItem(string key, T item)
{
    _map_lock.WaitOne();
    _list_lock.WaitOne();
    try
    {
        _list[_map[key]] = item;
        _map_lock.Set();
        _list_lock.Set();
        return true;
    } catch (Exception)
    {
        _map_lock.Set();
        _list_lock.Set();
        return false;
    }
}
```

Fig.5 A program with a dead-lock bug



```
Name  ConcurrentListBulkAppendTest

Test case 1 (ConcurrentListSetup) × 1
Pending time: 20ms

Test case 2 (ConcurrentListRandomAppendTest) ×1000
Pending time: 1s

Test case 3 (ConcurrentListShutDownTest) × 1
Pending time: 0ms
```

Test Case: ConcurrentListRandomAppendT

Repeat Times: 20

Pending Time: 5 ms

Fig.6 Workload design interface

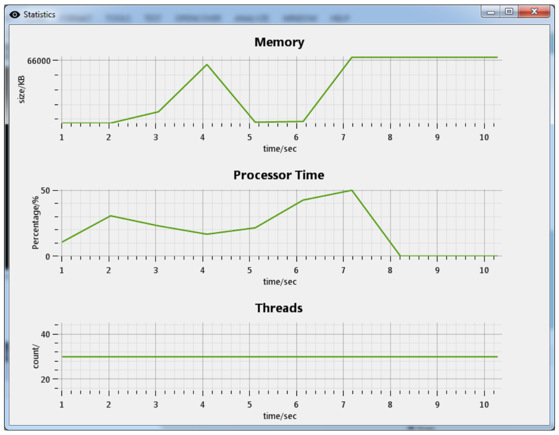
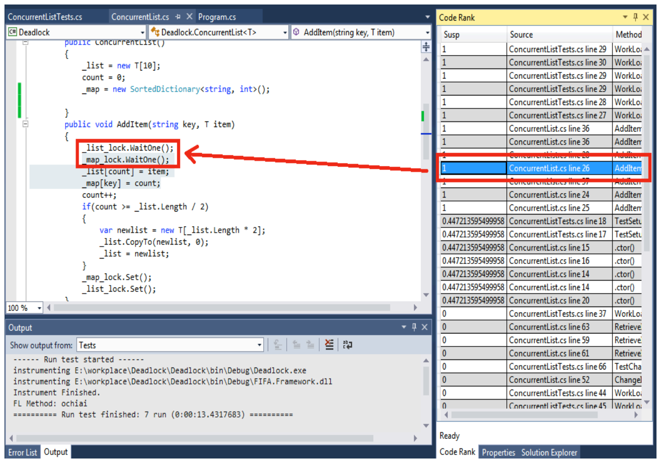


Fig. 7 Environmental factor monitoring interface



```
public ConcurrentList()
{
    _list = new T[10];
    count = 0;
    _map = new SortedDictionary<string, int>();

}
public void AddItem(string key, T item)
{
    _list_lock.WaitOne();
    _map_lock.WaitOne();
    _list[count] = item;
    _map[key] = count;
    count++;
    if(count >= _list.Length / 2)
    {
        var newlist = new T[_list.Length * 2];
        _list.CopyTo(newlist, 0);
        _list = newlist;
    }
    _map_lock.Set();
    _list_lock.Set();
```

Show output from: Tests

```
------ Run test started ------
instrumenting E:\workplace\Deadlock\Deadlock\bin\Debug\Deadlock.exe
instrumenting E:\workplace\Deadlock\Deadlock\bin\Debug\FIFA.Framework.dll
Instrument Finished.
FL Method: ochiai
========== Run test finished: 7 run (0:00:13.4317683) ==========
```

Susp	Source	Method
1	ConcurrentListTests.cs line 29	WorkLo
1	ConcurrentListTests.cs line 30	WorkLo
1	ConcurrentListTests.cs line 29	WorkLo
1	ConcurrentListTests.cs line 29	WorkLo
1	ConcurrentListTests.cs line 28	WorkLo
1	ConcurrentListTests.cs line 27	WorkLo
1	ConcurrentList.cs line 36	AddItem
1	ConcurrentList.cs line 36	AddItem
1	ConcurrentList.cs line 28	AddItem
1	ConcurrentList.cs line 26	AddItem
1	ConcurrentList.cs line 37	AddItem
1	ConcurrentList.cs line 24	AddItem
1	ConcurrentList.cs line 25	AddItem
0.447213595499958	ConcurrentListTests.cs line 18	TestSetu
0.447213595499958	ConcurrentListTests.cs line 17	TestSetu
0.447213595499958	ConcurrentList.cs line 15	.ctor()
0.447213595499958	ConcurrentList.cs line 16	.ctor()
0.447213595499958	ConcurrentList.cs line 14	.ctor()
0.447213595499958	ConcurrentList.cs line 14	.ctor()
0.447213595499958	ConcurrentList.cs line 20	.ctor()
0	ConcurrentListTests.cs line 37	WorkLo
0	ConcurrentList.cs line 63	Retrieve
0	ConcurrentList.cs line 59	Retrieve
0	ConcurrentList.cs line 61	Retrieve
0	ConcurrentListTests.cs line 66	TestCha
0	ConcurrentList.cs line 52	Change
0	ConcurrentListTests.cs line 44	WorkLo
0	ConcurrentListTests.cs line 45	WorkLo

Fig. 8 Code rank of a program with deadlock bug

In this example, the program implemented a concurrent list data structure (as shown in Fig. 5), which has a dead-lock bug. Thus, once under intensive writing operations, the program can get locked.

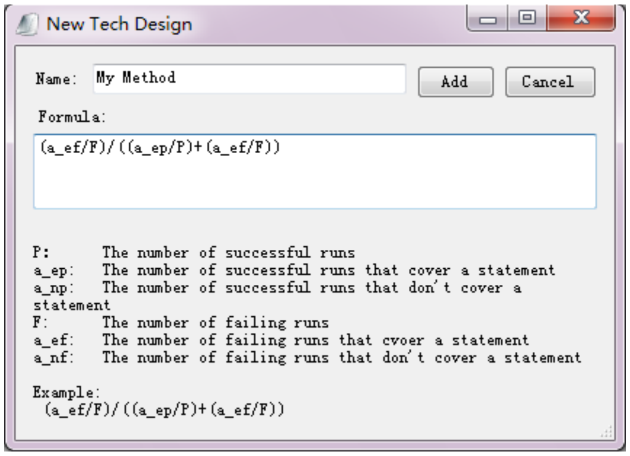
The developer first design several workloads for the program under test (as shown in Fig.6), and display all workloads in the test explore. After running tests, the developer can select and then view each test's environmental factors monitored by FLAVS (as shown in Fig.7). The developer notices that after about 8 seconds running, the processor time dropped to zero and did not increase again. It indicates a failure.

After marking the failure in FLAVS, the fault localization is activated. The FLAVS window showed the executed statements as well as their calculated suspiciousness degrees (as shown in Fig.8). The programmer checked the statements one by one and observed that the 10th statement is faulty. The programmer clicked on the statement in the FLAVS window and jumped to the source code page. The dead-lock bug is found.

If the developer attempts to design new fault localization technique, he can open the interface as shown in Fig.9 and design the formula according to the instructions. As discussed before, the formula was composed using six predefined variables [12].

V. Conclusion and Future Work

Debugging is always a time-consuming and tedious task, while no automatic fault localization work is widely adopted in practice. Based on comments and feedbacks in promoting dynamic fault localization in our experiences, we propose



```
Name:  My Method

Formula:
(a_ef/F)/((a_ep/P)+(a_ef/F))

P:      The number of successful runs
a_ep:   The number of successful runs that cover a statement
a_np:   The number of successful runs that don't cover a
statement
F:      The number of failing runs
a_ef:   The number of failing runs that cvoer a statement
a_nf:   The number of failing runs that don't cover a statement

Example:
 (a_ef/F)/((a_ep/P)+(a_ef/F))
```

Fig.9 New fault localization technique design interface

FLAVS, which is to transfer the approaches of dynamic fault localization to practical tools. FLAVS seamlessly integrates the dynamic fault localization with a popular IDE - Microsoft Visual Studio. It in background conducts instrumentation, gathers coverage information, collects environmental factors and calculates suspicious degrees for basic blocks. The designed test cases are fed to a test explorer without any interruption to user activities. The tool is useful for developers working with Microsoft Visual Studio, and is also helpful for software testing researchers in debugging complex bugs like Mandelbugs.

Our future work mainly includes two aspects. First, our tool currently only supports statement-, branch-, and function-level program units for suspiciousness assessing and ranking. In future work, we will extend the instrumentation framework of our tool so that the users can use path profile and data-flow profile to perform fault localization. Second, currently, only C# projects are supported. We will extend our tool for the user using other program languages, such as C++, Basic and etc. Third, we will extend the tool for other popular developing environment, such as Borland C++, JBuilder, and Delphi.

ACKNOWLEDGMENT

The work is supported by Beijing Higher Education Young Elite Teacher Project (No. YETP1072).

REFERENCES

[1] R. Abreu, J. Cunha, J. P. Fermande, P. Martins, A. Perez, and J. Saraiva, "FaultySheet Detective: When Smells Meet Fault Localization," ICSME 2014, pp. 625-628.

[2] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip. "Crisp - a fault localizatoin tool for Java programs," ICSE 2007, pp.775-779.

[3] K. A. George, A. Podgurski, and M. J. Harrold. "Mitigrating the confounding effects of program dependences for effective fault localizatoin," ESEC 2011/FSE-19, pp. 146-156.

[4] M. Grottke, K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," IEEE Comput., vol. 40, no. 2, pp. 107-109, 2007.

[5] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "VIDA: Visual interactive debugging", ICSE 2009, pp. 583-586.

[6] T. Janssen, R. Abreu, and A. J. C. van Gemund, Zoltar: A Toolset for Automatic Fault Localization, ASE 2009, pp.662-664.

[7] J. A. Jones and M. J. Harrold. "Empirical evaluation of the Tarantula automatic fault-localization technique," ASE 2005, pp.273-282.

[8] M. Jose, and R. Majumdar, "Bug-Assist: assisting fault localization in ANSI-C programs, Computer Aided Verification," vol. 6806, pp. 504-509, 2011.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. "Bug isolation via remote program sampling," PLDI 2003, pp. 141-154.

[10] C. Liu, L. Fei, X. Yan, S. P. Midkiff, and J. Han. "Statistical debugging: a hypothesis testing-based approach,". IEEE Transactions On Software Engineering, vol. 32, no. 10, pp. 831-848, 2006.

[11] Math.NET Numerics, http://numerics.mathdotnet.com. Last accessed on June 30th, 2014.

[12] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, no. 3, 11, 2011.

[13] S. Park, R. W. Vuduc, and M.J. Harrold, Falcon: fault localization in concurrent programs, ICSE 2010, pp. 245-254.

[14] I. Vessey. "Expertise in debugging computer programs: an analysis of the content of verbal protocols," IEEE Transactions on Systems, Man and Cybernetics, vol. 16, no. 5, pp. 621-637, 2007.

[15] W. Eric Wong, Y. Qi, L. Zhao, K. Y. Cai. "Effective Fault Localization using Code Coverage," COMPASAC 2007, pp. 449-456 .

[16] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 22, no. 4, 31, 2014.

[17] X. Zhang, N. Gupta, and R. Gupta. "Locating faults through automated predicate switching," ICSE 2006, pp. 271-281.

[18] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. "Capturing propagation of infected program states," ESEC 2009/FSE-17, pp.43-52.