Regression Testing: A Spectrum-based Approach

Shailesh Tiwari
CSED
MNNIT Allahabad

K.K Mishra
CSED
MNNIT Allahabad

A.K. Misra
CSED
MNNIT Allahabad

ABSTRACT
Regression testing involves re-run of all test suite or selective run of a sub-set of existing test cases on the modified version of program to reveal the regression faults due to changes in code and use of these non obsolete test cases from pre-existing test suite to explore and eradicate regression faults. This paper addresses the fundamental limitations of conventional regression testing approach and presents a spectrum-based fault localization strategy by which the stated limitations are resolved in effective manner. Spectrum-based fault localization strategy utilizes various program spectra to identify the behavioral differences between old and new version of the program under test. This comparison is also useful in pinpointing the cause of failures or errors and presence of difference in program spectra may indicate those test cases for which the construction of expected output or oracle or specification is not needed. The present approach can identify and localize the faults effectively and also identify those test cases from pre-existing test suite available for existing program that exercise the changed behavior of the modified code. Further the developer can easily identify whether the differences recorded in modified version of code is due to regression faults or due to changes made in the code.

Keywords
Regression Testing, Fault Localization, Program Spectrum, Behavioral Regression Testing.

1. INTRODUCTION
Software systems are maintained by developers by doing regression test periodically in hope to find errors caused by changes and provide confidence that modifications made in the software are correct. Developers often create an initial test suite and then reuse it for regression testing [1]. These initial test suites are generally saved by the developers in order to reuse these test suites in regression testing as their software evolves. This reuse of test suites is pervasive in software industries [2].

A wide variety of research topics on regression testing are given in the literature. Some focus on test environments and automation of regression testing process [4, 8, 9] while others investigate automated capture playback mechanisms and test suite management [10]. Various algorithms are also given in literature that addresses test suite management [6, 7]. Another algorithm was presented in [5] that constructs a reduce size version of modified program for use in regression testing.

Regression testing tasks constitute a significant percentage of the costs of software testing as cost always increases when modifications are made in later stages of software development. A major difference between regression testing and development testing is that during regression testing a well established test suite is available for reuse. A basic strategy for regression testing is retest-all strategy i.e. retest all the test of test suite but it may consume excessive time and resources which may lead to increase in cost. On the other side, regression test selection strategy reduces the time required to retest a modified program by selecting some subset of the existing test suite. Therefore the methods that reduce the cost of regression testing tasks are always valuable. Most recent researches in regression testing concerns selective retest techniques. Selective regression testing approaches are described in [3]. But there are some fundamental limitations with the conventional test selection techniques discussed in section II.

When software system evolves, regression testing is done to compare the behavior of the modified version to the behavior of its previous version. This comparison is also useful in pinpointing the cause of failures or errors [11]. The behavior of the program can be measured by characterizing the program spectrum. In literature, the program spectra can be characterized in many ways: path profiling, path spectra, node spectra, edge spectra, branch spectra, complete path spectrum, data dependence spectra, output spectrum, execution trace spectrum, value spectra, block spectra etc. [12, 13]. The analysis of program spectra provides an understanding to testing and maintenance tasks. Presence of difference in program spectra may also indicate those test cases for which the construction of expected output or oracles or specifications is not needed. This may also help developers to locate the faults in the program [11].

Various program spectrum-based fault localization approaches has been discussed in literature [11, 14] in which program spectra are used to capture the execution information from the successful and failed test cases. The behavior of the program is characterizes in terms of program spectra from different executions of the test cases on the source code. Program spectra are also used to quantify the quality of fault localization.

This paper proposes an approach for program spectrum-based fault localization in regression testing that identify the behavioral differences between two versions of programs, pinpoint the faults, and identify those test cases that exercise the changed behavior of the modified version of the program.

2. PROBLEM ANALYSIS

2.1 Regression Testing
Regression testing is considered as a test cycle by which a new version of program is tested to ensure that newly added or modified code behaves correctly as well as unchanged previous version of code continues to behave correctly. It is also referred as 'Program Revalidation'. The behavior of the old and new version of program under test (PUT) has been compared by running same test on these two versions of program. If they produce different output on same test then deviation in behavior has been exposed. The developer can easily detect the faults by considering the behavioral deviations between old and new version of program.

Regression testing is a process to uncover errors by partially retesting a modified program. Regression testing is done after modification is made in the implemented program. This can be done by rerunning the existing test suites against the modified code to determine whether the changes affects anything that worked properly prior to the change or writing new test cases where necessary. Adequate coverage should be primary consideration when conducting regression tests [15].

For simplification:

Let P be the program and P' be the modified program; let T be the set of test cases for P then T′ is selected from T that is subset of T for executing P′, establishing T′ correctness with respect to P′, if necessary, create T″ and execute T″ on P′, establishing T″ correctness with respect to P′, if necessary, create T‴ and execute T‴ on P′, establishing T‴ correctness with respect to P′. Each of these steps is involved with some problems of selective retest technique: Regression test selection problem, Coverage identification problem, Test suite execution problem and Test suite maintenance problem [15].

Regression testing can be distinguished in two phases: a preliminary phase and a critical phase. In preliminary phase, test history and code analysis information is gathered to enhance and correct the software while in critical phase regression testing is done. If the information gathered in preliminary phase is used in critical phase then the cost of critical phase of regression testing can be reduced [15].

In conventional regression testing approach, developers generally re-run all test suite or selectively run a set of existing test cases i.e. T′ from existing test suite on the modified version P'. After executing T′ over P', developer may reveal the regression faults due to changes in code and use these non obsolete test cases from pre-existing test suite to explore and eradicate regression faults [29]

Conventional regression testing approaches that depend only on pre-existing test suite have some fundamental limitations derived from test set problem [12] and oracle problem [12, 13]:

- If the pre existing test suite i.e. T is lacking in quality then regression testing may become ineffective as it completely rely on the pre existing test suite T [18].
- If pre existing test suite is manually generated then it is very difficult and time consuming to achieve high structural coverage of non trivial program [18].
- Be short of test cases that exercised a changed behavior of the existing and changed version of program. [29]
- Be short of a test oracle or comparator that can classify changed behavior of the existing and changed version of the program. [29].
- If, new test case generation process is required then how it will be initiated.

Consider a program that has changed from P to P', and having a test suite T created for and already executed on P, a regression testing process typically involves [19, 20, 21]:

I. Maintaining T to get T': identifying redundant and obsolete test cases and repairing or discarding them.

II. Optimizing T' to get T": selecting a subset of test cases and prioritizing the selected test cases.

III. Running T" on P': checking the results, and identifying failures.

IV. Creating a new test suite T"', if needed, to test P'.

V. Running T"' on P': checking the results and identifying potential failures.

VI. Aggregating T" and T"' to get T"' and saving it for future.

We have the following observations from the above regression testing process:

O1. If the pre existing test suite is lacking in quality then regression testing may become ineffective as it completely relies on the pre existing test suite T.

O2. In that case there is no need to remove obsolete and redundant test cases from the test suite initially.

O3. If somehow, pre-existing test suite is capable then it may be short of test cases that exercise the changed behavior of the code as test result is completely relies on deviation of outputs.

O4. How failures are identified and localized if the program is non-trivial?

After concretely analyzing these observations, we concentrate on latest existing work on spectrum-based fault localization and its implementation in regression testing. Program spectrum-based fault localization approaches are heuristic approaches that utilize various program spectra acquired dynamically from software testing. It is simple, easy to use, and effective and also has become a promising technique.

2.2 Program Spectra

The term program spectra come from path profiling in which distribution of the paths derived from run of the program. A program spectrum tracks the program's run-time behavior by recording the execution information of a program in certain aspects such as execution of conditional branches or execution of intra-procedural paths or execution of def-use pairs etc. [13, 22]. Various program spectra have been given in literature [11, 22] used as a component to quantify the quality of fault localization approach by deriving its correlation with the program's behavior. We consider ten distinct types of program spectra that are widely used in literature.

Table 1 Types of program spectra

	Name	**Description**
BHS	Branch Hit Spectra	Conditional Branches that are executed
BCS	Branch Count Spectra	Number of times each conditional branch is executed
CPS	Complete Path Spectra	Complete path that is executed
PHS	Path Hit Spectra	Loop-free path that is executed
PCS	Path Count Spectra	Number of times each loop-free path is executed
DHS	Data-dependence Hit Spectra	Definition-use pairs that are executed
DCS	Data-dependence Count Spectra	Number of times each definition-use pair is executed

OPS	Output Spectra	Output that is produced
ETS	Execution Trace Spectra	Execution trace sequence that is produced
DVS	Data Value Spectra	Values of variables in the execution

Initially, path profiling [12, 24] is used as path spectra to track intra-procedural paths in the program. Harrold et al. [11] proposed first nine kind of program spectra listed in Table 3.1. These spectra are constructed on the basis of node, edge, output and execution trace etc. They also define the empirical view of the relationships among these nine types of spectra. Out of these nine types of spectra, output spectra can be referred as semantic spectra, which consist of the outputs (values) produced by the program execution and rest of the spectra can be referred as syntactic spectra, which consist of signature of structural entities exercised by the program execution [25]. Bowring et al. [26] suggest new spectrum called data value spectra that track the transition of the values in variables as program executes. We next describe these spectra in brief.

Branch Hit Spectra: If, for each conditional branch in program P, the branch hit spectra (BHS) merely record whether that branch is exercised or not. It records the conditional branches that are covered by the test execution.

Branch Count Spectra: If, for each conditional branch in program P, branch count spectra (BCS) merely record number of times that branch is exercised by the test execution.

Path Hit Spectra: If, for each loop-free, intra-procedural path in control flow graph G of program P, path hit spectra merely record whether that path is exercised or not. It records the loop-free, intra-procedural paths that are covered by the test execution.

Path Count Spectra: If, for each loop-free intra-procedural path in control flow graph G of program P, path count spectra merely record the number of times that path is exercised by the test execution.

Complete Path Spectra: records the complete path traversed by the test execution.

Data-dependence Hit Spectra: If, for each definition-use pair in program P, the data-dependence hit spectra merely record whether that definition-use pair is exercised or not. It records the definition-use pairs that are covered by the test execution.

Data-dependence Count Spectra: If, for each definition-use pair in program P, the data-dependence count spectra merely record the number of times that definition-use pair is exercised by the test execution.

Output Spectra: It records the outputs (values) produced by the test executions.

Execution-trace Spectrum: It records the sequence of each program statement traversed by the test execution. Complete path spectra (CPS) and execution-trace spectrum (ETS) may appear to be similar as both records the complete control flow path through program P. The difference is that execution-trace spectrum (ETS) records the actual instructions executed along the path whereas complete path spectra (CPS) doesn't.

Data Value Spectra: It records the transition of the values of variable [databin]. When a test case is executed, the transition sequence of values of variables are recorded, which is one of the data value spectra representation.

Harrold et al. [EIPS] empirically investigate the co relationship between program spectra differences and exposure of program failure behavior. They utilize two measures to investigate and quantify the relationship exists between inputs that cause program P and faulty program P' to produce different program spectra and inputs that cause program P and faulty program P' to produce different failure behavior. These measures are:

Given program P, faulty version of program P' and an input set I for program P. Let FR(P, P', I) be the set of input test cases in I that cause program P' to fail. For such spectra S, let SR(P, P', I) be the set of inputs in I that produce spectra differences on program P and faulty version of program P'.

Degree of Imprecision: For spectra type S, any test input *i* in 'I' that is in SR(P, P', I) but not in FR(P, P', I) exhibits a spectra difference under S that is not correlated with a failure. For such a test input *i*, spectra comparison is 'imprecise' and the degree of imprecision of S-spectra with respect to P, P', and I is given by the equation:

$$\frac{|\mathrm{SR}(P, P', I) - \mathrm{FR}(P, P', I)|}{|\ \mathrm{SR}(P, P', I)\ |} \quad (1)$$

Degree of Unsafety: For spectra type S, any input *i* in 'I' that is in FR(P, P', I) but not in SR(P, P', I) exhibits a failure that is not correlated with a spectra difference of type S. For such test input *i*, spectra difference is 'unsafe' and degree of unsafety of S-spectra difference with respect to P, P' and I is given by the equation:

$$\frac{|\ \mathrm{FR}(P, P', I) - \mathrm{SR}(P, P', I)\ |}{|\ \mathrm{FR}(P, P', I)\ |} \quad (2)$$

Peifung Hu [27] extends the degree of imprecision and degree of unsafety of S-spectra in terms of successful and failed test inputs by giving following equations:

Let SR(P, P', I_S) be the set of test inputs in I_S that produce spectra difference on program P and faulty version of program P' and SR(P, P', I_F) be the set of test inputs in I_F that produce spectra difference on program P and faulty version of program P'. Now, the equation (1) can be written as:

$$\frac{|\ \mathrm{SR}(P, P', I_S)|}{|\ \mathrm{SR}(P, P', I_S) + \mathrm{SR}(P, P', I_F)\ |} \quad (3)$$

Similarly, equation (2) can be written as:

$$\frac{|I_F\ | - |\mathrm{SR}(P, P', I_F)|}{|(IF)\ |} \quad (4)$$

On the empirical investigation of these equations, Peifung Hu [27] argues that a good spectrum which can be used for fault localization should have: low degree of imprecision and unsafety.

3. PROPOSED APPROACH

A novel spectrum-based fault localization approach has been proposed that identify behavioral differences between existing and modified version of program. Goal of proposed approach is to accurately identify behavioral differences between two

versions of program on the basis of program spectra by means of static analysis of source code. This may also help developers to localize the faults in a program. Presence of difference in program spectra may also indicate those test cases for which the construction of expected output or oracles or specifications is not needed. Test cases that reflect the spectra differences are considered as fault or modification revealing test cases as they exercised the changed behavior of the source code. They also have greater fault detection capability as compared to other test cases as they capture changes in the source code.

3.1 Why these Spectra?

As discussed in previous section of this chapter, a good spectrum which can be used for fault localization approach should have: low degree of imprecision and unsafety. But, it is impractical to calculate the degree of imprecision and unsafety for each spectra on a specific program with its test suite as we do not have the correct version of existing program P in practical debugging situation.

To use good spectra in our fault localization approach, we conduct an experiment on *printtokens*, *printtokens2*, *replace*, *schedule*, *schedule2*, *tcas*, *and totinfo* with 25,542 input universe size, taken from *Software-artifact Infrastructure Repository* [28]. Table 2 shows the evaluated degree of imprecision and degree of unsafety of each spectrum.

Table 2 Evaluated degree of imprecision and unsafety

	Name	Degree of Imprecision	Degree of Unsafety
BHS	Branch Hit Spectra	16%	13%
BCS	Branch Count Spectra	19%	8%
CPS	Complete Path Spectra	24%	8%
PHS	Path Hit Spectra	6%	14%
PCS	Path Count Spectra	15%	29%
DHS	Data-dependence Hit Spectra	43%	22%
DCS	Data-dependence Count Spectra	44%	18%
OPS	Output Spectra	0%	0%
ETS	Execution Trace Spectra	0 %	92%
DVS	Data Value Spectra	0 %	95%

On the basis of these experimental results, following spectrum are selected for proposed fault localization approach in regression testing:

1) Branch Count Spectra (BCS): It has higher possibility of having failed test cases having a different spectra on program P and modified version of program P' as it have lower degree of unsafety as compared to other spectra whereas it has lower degree of imprecision as compared to other spectra except BHS and PHS. Therefore, BCS is selected.

2) Branch Hit Spectra (BHS): It has higher possibility of a spectra difference correlated to a regression failure as it have lower degree of imprecision as compared to other spectra except PHS whereas it has lower degree of unsafety as compared to other spectra except BCS, PHS, PCS, and CPS. Therefore, BHS is considered for the proposed approach.

3) Path Hit Spectra (PHS): It has higher possibility of a spectra difference correlated to a regression failure as it have lower degree of imprecision as compared to other spectra whereas it has lower degree of unsafety as compared to other spectra except BCS, PCS, and CPS. We will select path hit spectra (PHS) rather than path count spectra (PCS) and complete path spectra (CPS) because it has quite lower degree of imprecision among them and in terms of degrees of imprecision and unsafety the CPS and PCS display nearly identical behavior. Other reasons are: it is cost consuming to collect all path spectra and it is also difficult to map path spectra into fault locations in practical situation. Therefore, PHS is considered rather than PCS and CPS.

4) Execution-trace Spectrum (ETS): It is selected because of its relationship with regression testing discussed in controlled regression testing [analyzing regression test], assuming the tests that produce different execution-trace spectra constitute a safe approximation. Among all spectra, ETS is safe because for all test inputs that reveal faults also reveal ETS differences. Every failed test case in ETS having a different spectra on program P and modified version of program P'. The degree of imprecision of ETS is higher as compared to other spectra except DVS because ETS subsumes all the spectra.

Output spectra (OPS) are not selected because we cannot identify the behavior of the two versions of program as it only provides the differences in outputs. Data value spectra (DVS) are not considered as it has the highest degree of imprecision and the lowest degree of unsafety because of its highest degree of imprecision and it is also expensive and difficult to capture the values of variables. Data-dependence hit (DHS) and data-dependence count (DCS) spectra are not considered due to their higher degree of imprecision and unsafety.

3.2 Program Spectra Comparisons

Consider an example:

Let P be the existing version of sample program and P' be the modified version of sample program written in 'C^{++}, shown in figure 1.

These sample programs simply swap and increment the values of variables by using conditional statements and loop. Program P contains 11 statement blocks and 27 lines of code. Program P' is the modified version of P, contains 13 statement blocks and 35 lines of code. The control flow of the programs is discussed by if-else statements. While( ) loop may be used as multiple loop executions in a path.

The programs are given as:

```
#include<iostream.h>
void main()
int a,b,c,n;
cin>>a>>b;
if (a<b)
{
c = a;
else
{
c = b;
}
n = c;
while (n<8)
{
If (b>c)
{
c = 2;
}
else
{
n = n+c+7;
}
n = n+1;
}
cout<<a<<b<<n;
}
```

```
#include<iostream.h>
void main()
int a,b,c,n;
cin>>a>>b;
if (a<b)
{
c = a;
else
{
c = b;
}
n = c;
while (n< =8)
{
If (b>c)
{
c = 2;
}
else
n = n+c+7;
if (n%7==0)
{
c = c+2;
}
else
{
c = c-1;
}
}
n= n+1;
}
cout<<a<<b<<n;
}
```

Figure 1 Program P and modified program P'

Now, the control flow graphs (CFG) are constructed for existing sample program P and P'. Constructed CFG's are shown in figure 2:

Program spectra for existing sample program P and modified version of program P' on 'I' are recorded and represented in the given Table 3 and Table 4, respectively:

The comparison of spectra between old and new version of program is depend on the following assumption:

Branch hit spectra (BHS), branch count spectra (BCS), path hit spectra (PHS), and execution-trace spectrum of both existing and modified version of program is compared on the basis, whether they are lexicographically equivalent or not? Two text strings are lexicographically equivalent if their text (ignoring extra white space characters when not contained in character constants) is identical ([19]). Assume that the branch hit spectra (BHS) of existing program P is BHS(P) and for modified version of the program P' is BHS(P'). BHS(P') is lexicographically equivalent to BHS(P) if and only if the sequence of conditional branches exercised as program executes are identical. If BHS(P') and BHS(P)are not lexicographically equivalent, it indicates the behavioral difference between existing and modified version of program.

The comparison of spectra between old and new version of program is depend on the following assumption:

Branch hit spectra (BHS), branch count spectra (BCS), path hit spectra (PHS), and execution-trace spectrum of both existing and modified version of program is compared on the basis, whether they are lexicographically equivalent or not?

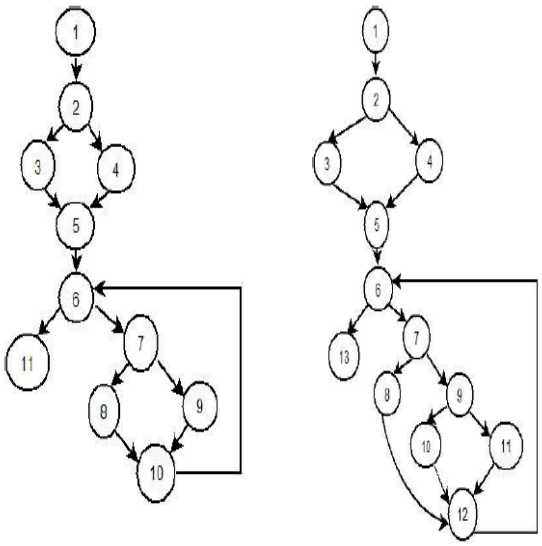

Figure 2 CFG for P and P'

Two text strings are lexicographically equivalent if their text (ignoring extra white space characters when not contained in character constants) is identical ([19]). Assume that the branch hit spectra (BHS) of existing program P is BHS(P) and for modified version of the program P' is BHS(P'). BHS(P') is lexicographically equivalent to BHS(P) if and only if the sequence of conditional branches exercised as program executes are identical. If BHS(P') and BHS(P)are not lexicographically equivalent, it indicates the behavioral difference between existing and modified version of program.

3.3 Spectrum-based Fault Localization (SBFL) Information

For these given program P and P', the behavioral differences between them are arranged as following information that is required for spectrum based fault localization (SBFL).

For the existing program P and modified program P', four spectra are computed i.e. Branch hit spectra (BHS), branch count spectra (BCS), path hit spectra (PHS), and execution-trace spectra (ETS).

For program P:

$P(I)_{Spectra} = \{BHS_P(I), BCS_P(I), PHS_P(I), ETS_P(I)\}$

For program P':

$P'(I)_{Spectra} = \{BHS_{P'}(I), BCS_{P'}(I), PHS_{P'}(I), ETS_{P'}(I)\}$

Where 'I' is the input test set available for the existing program P. 'I' contains both successful and failed test cases represented as 'I_S' and 'I_F', respectively.

A vector 'T_{case}' and a matrix 'S_{mat}' is created to track the compared result for each spectrum. Here, spectra recorded for modified program i.e. $P'(I)_{Spectra}$ is compared with the recorded spectra for existing program i.e. $P(I)_{Spectra}$ and behavioral differences are identified. If spectra is lexicographically equivalent to the corresponding spectra for existing program P for a given test case then it indicates 1 otherwise 0. For simplicity, 1 represents no behavioral difference and 0 shows that behavioral difference is recorded

in the given test case. Here, if ETS reflects the behavioral differences but no other spectra reflect the same, it means that there must be behavior differences in other spectra that are excluded in our study. For all test cases that reveal behavioral differences, it must be reflected in ETS.

Figure 4 shows the information required by proposed spectrum-based fault localization approach. This information in the form of vector and matrix identifies the test cases that capture the behavioral differences.

$$T_{CASE} = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ : \\ : \\ : \\ T_n \end{pmatrix} \quad S_{MAT} = \begin{pmatrix} BHS & BCS & PHS & ETS \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ - & - & - & - \\ - & - & - & - \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 3 SBFL information

Now, another vector $BDA_{Spectum}$ is created that analyze the behavioral differences obtained by the program spectra by test execution on modified program P'. This vector uses AND (∧) operation to analyze the obtained program spectra and the value obtained by AND operation on BHS, BCS, PHS, and ETS on corresponding test case reflects the behavioral differences in modified program P' as compared to its existing version of program P. If the value obtained by $BDA_{Spectum}$ after AND operation on execution of any test case is 1, it means that there is no behavioral difference recorded by that test case, otherwise, in case of 0, behavioral differences are recorded.

The behavioral differences analyzer $BDA_{Spectum}$ is shown in figure 4. Behavioral differences analyzer $BDA_{Spectum}$ reflect the values corresponding to each test case. For example, here the value corresponding to T_1 is 1, reflects that there is no behavioral difference is recorded in any type of spectra or in other words, no changes or regression faults are found in that part of modified version of code executed by T_1. In case of T_2, and T_3, the value is 0 reflects that there must be some changes or regression faults are present in that part of modified version of code executed by T_2, and T_3.

Now, the code with analysis is given to the developer to check whether the changes in behavior of modified version of program is due to regression faults or due to changes made in the code? If regression fault is present then these faults are removed without execution of test cases and there is no need to write or check oracles or specifications for them. If behavior of the program is changed due to changes made in the modified version of code then new test cases are need to be generated.

Now, the code with analysis is given to the developer to check whether the changes in behavior of modified version of program is due to regression faults or due to changes made in the code? If regression fault is present then these faults are removed without execution of test cases and there is no need to write or check oracles or specifications for them. If behavior of the program is changed due to changes made in the modified version of code then new test cases are need to be generated.

$$BDA_{Spectum} = \begin{matrix} BHS \wedge BCS \wedge PHS \wedge ETS \\ \begin{pmatrix} T_1 & 1 \\ T_2 & 0 \\ T_3 & 0 \\ : & : \\ : & : \\ : & : \\ T_n & 1 \end{pmatrix} \end{matrix}$$

Figure 4 Behavioral differences analyzer ($BDA_{Spectum}$)

Now, the code with analysis is given to the developer to check whether the changes in behavior of modified version of program is due to regression faults or due to changes made in the code? If regression fault is present then these faults are removed without execution of test cases and there is no need to write or check oracles or specifications for them. If behavior of the program is changed due to changes made in the modified version of code then new test cases are need to be generated.

The most significant contribution of the proposed approach is that it easily identifies those test cases from the pre-existing test suite available for the initial program, which executes the changed behavior of the modified version of program. These test cases are considered as modification-traversing test cases. A test case t_i that belongs to input test suite 'I' is modification-traversing for modified version of program P' if and only if the value of behavioral differences analyzer $BDA_{Spectum}$ is 0.

For the purpose of regression test selection, we want to identify all tests T that reveal faults in P' — the fault-revealing tests. An approach that selects every fault-revealing test in T is safe. There is no effective procedure that identifies the fault-revealing tests in T [3]. However, under controlled regression testing, the modification-traversing tests are a superset of the fault-revealing tests [3]. Thus, for controlled regression testing, a regression test selection approach that selects all modification-traversing tests is safe. It means that a test case is safe and fault-revealing if it is modification-traversing [19]. Therefore, the proposed approach is also a safe regression test identification approach that identifies the modification-traversing as well as fault revealing test cases.

4. EXPERIMENTAL RESULTS

We developed a program in C that compare the two strings collected as the program spectra on existing program P and modified program P' by test execution. These string comparisons for each program spectrum to the corresponding test cases are stored in the form of values 1 or 0 in a matrix i.e. S_{MAT}. gcov tool [gcov] is used to compute the selected program spectra differences on two versions of program. Now, a vector $BDA_{Spectum}$ is created that stores the value in the form of 1 or 0 after performing the AND (∧) operation on the values stored in matrix S_{MAT}. The value of $BDA_{Spectum}$ reflects that for each test case, whether or not, the behavioral differences are recorded on the modified version of program P'.

We used six C programs as subjects in the experiment. First program is simple program to identify the type of the triangle

and other five programs with faulty versions and a set of test cases are taken from Software-artifact Infrastructure Repository (SIR) [SIR]. The faulty version of a program can be created by manually seeding the faults in the existing version of program to make it differs from the existing program by one to seven lines of code.

Table 5 shows the subject program name, description, lines of code, number of faulty version created, number of test cases in input test suite 'I' that is available for existing program P, and number of test cases in input test suite 'I' that reflects the spectra differences on modified version of program P'. The creation of faulty version of program simulates the scenario of introducing regression faults into the subject programs during modifications.

Table 5 Experimental results

Program	LOC	Faulty Version	Tests	Number of Tests reflects spectra differences
triangle	26	1	28	7
printtok2	483	10	4115	532
replace	516	12	5542	2110
schedule	299	9	2650	1214
tcas	138	9	1608	465
totinfo	346	6	1052	243

From results, we observed that those test cases are identified that reflect the behavioral differences between two version of program by means of comparing selected program spectra, statically. These test cases are modification-traversing test cases and have better fault detection capability over the rest of the test cases present in the existing test suite as they exercised the changed behavior of the modified version of the program.

5. CONCLUSION

In this paper, we investigate the process of spectrum-based fault localization approaches and proposed an approach for regression testing to identify four issues: selection of suitable program spectra for fault localization in regression testing, a process to identify the behavioral differences on two versions of program by test execution, identification of type of fault and pinpointing of faults, identification of modification-traversing and fault revealing test cases.

We compare ten program spectra given in literature in terms of degree of imprecision and degree of unsafety to find out the best suitable program spectra for the proposed approach. From experimental results, four program spectra are selected: branch hit spectra (BHS), branch count spectra (BCS), path hit spectra (PHS), and execution-trace spectra (ETS). These program spectra have been utilized in the proposed fault localization approach for regression testing.

Branch hit spectra (BHS), branch count spectra (BCS), path hit spectra (PHS), and execution-trace spectra (ETS) are used to find out the behavioral differences on two versions of programs. The proposed approach effectively identifies that part of the code which consists of changes or regression faults. If regression fault is present then these faults are removed without execution of test cases and there is no need to write or check oracles or specifications for them. If behavior of the program is changed due to changes made in the modified version of code then new test cases are need to be generated. The proposed approach can effectively localize the faults in terms of branch, path, and execution spectra deviations.

This paper provides an understanding for the process program spectrum-based of fault localization in regression testing by identifying those test cases from the pre-existing test suite available for the existing program that exercised the changed behavior of the modified program.

Table 3 Evaluated spectra for existing program P on 'I'

Input	BHS	BCS	PHS	ETS
{-1, 3}	{2, 3}, {6, 7}, {7, 8}, {6, 11}	{2, 3}, {6, 7}^{10}, {7, 8}^{9}, {6, 11}	{1, 2, 3, 5, 6, 7, 8, 10, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < 8)) ^{10}, {(if (b > c), (c = 2;), (n = n + 1;)}^{9}, (cout << a << b << n;) }]
{2, 10}	{(2, 3), (6, 7), (7, 8), (6, 11)}	{(2, 3), (6, 7)^{7}, (7, 8)^{6}, (6, 11)}	{1, 2, 3, 5, 6, 7, 8, 10, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < 8)) ^{7}, {(if (b > c), (c = 2;), (n = n + 1;)}^{6}, (cout << a << b << n;) }]
{6, 2}	{2, 4}, {6, 7}, {7, 9}, {6, 11}	{2, 4}, {6, 7}, {7, 9}, {6, 11}	{1, 2, 4, 5, 6, 7, 9, 10, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), else {(c=b;), (n = c;), {(while (n < 8), {(if (b > c), else { (n=n+c+7;), (n = n + 1;)}^{6}, (cout << a << b << n;) }]

{2, 13}	{(2, 3), (6, 7), (7, 8), (6, 11)}	{(2, 3), (6, 7)^{7}, (7, 8)^{6}, (6, 11)}	{1, 2, 3, 5, 6, 7, 8, 10, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < 8)) ^{7}, {(if (b > c), (c = 2;), (n = n + 1;)}^{6}, (cout << a << b << n;) }]
{10, 14}	{2, 3}, {6, 11}	{2, 3}, {6, 11}	{1, 2, 3, 5, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), else {(c=b;), (n = c;), {(while (n < 8), (cout << a << b << n;)]
{7, -2}	{2,4}, {6,7}, {7,9}, {6,11}	{2,4}, {6,7}^{3}, {7,9}^{2}, {6,11}	{1, 2, 3, 5, 6, 7, 9, 10, 6, 11}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), else {(c=b;), (n = c;), {(while (n < 8)^{3}, {(if (b > c), else { (n=n+c+7;), (n = n + 1;)}^{2}, (cout << a << b << n;) }]

Table 4 Evaluated spectra for modified program P' on 'I'

Input	BHS	BCS	PHS	ETS
{-1, 3}	{2,3}, {6,7}, {7,8} , {6,13}	{2,3}, {6,7}^{11}, {7,8}^{10}, {6,13}	{1,2,3,5,6,7,8,12,6,13}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < =8))^{11}, {if (b > c){(c = 2);}, (n = n + 1);}^{10}, cout << a << b << n;}]
{2,10}	{2,3}, {6,7}, {7,8}, {6,13}	{2,3}, {6,7}^{8}, {7,8}^{7}, {6,13}	{1,2,3,5,6,7,8,12,6,13}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < =8))^{8}, {if (b > c){(c = 2);}, (n = n + 1);}^{7}, cout << a << b << n;}]
{6,2}	{2,4}, {6,7}, {7,9}, {9,11}, {6,13}	{2,4}, {6,7}^{2}, {7,9}, {9,11}, {6,13}	{1,2,4,5,6,7,9,11,12,6,13 }	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), else {(c=b;), (n = c;), {(while (n < =8)^{2}, {(if (b > c), else {(n = n + c +7;), if (n % 7 == 0), else{(c = c – 1;)}}, (n = n + 1;)} cout << a << b << n;}]
{2,13}	{2,3}, {6,7}, {7,8}, {6,13}	{2,3}, {6,7}^{8}, {7,8}^{7}, {6,13}	{1,2,3,5,6,7,8,12,6,13}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < =8))^{8}, {if (b > c){(c = 2);}, (n = n + 1);}^{7}, cout << a << b << n;}]
{10,14}	{2,3}, {6,13}	{2,3}, {6,13}	{1,2,3,5,6,13}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), (c = a;), (n = c;), {(while (n < =8)), cout << a << b << n;}]
{7,-2}	{2,4}, {6,7}, {7,9}, {9,11}, {6,13}	{2,4}, {6,7}^{3} , {7,9}^{2}, {9,11}, {6,13}	{1,2,4,5,6,7,9, 11,12,6,13}	[(int a, b, c, n;), (cin >> a >> b;), (if (a < b)), else {(c=b;), (n = c;), {(while (n < =8)^{3}, {(if (b > c), else {(n = n + c +7;), if (n % 7 == 0), else{(c = c – 1;)}}, (n = n + 1;)}^{2} cout << a << b << n;}]

6. REFERENCES

[1] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. "An empirical study of regression test selection techniques", ACM Trans. Softw. Eng. Methodology. 10, 2 (April 2001).

[2] J.C. Munson and T.M. Khoshgoftaar, "The detection of Fault Prone Programs", IEEE Trans. Software Eng. Vol. 18, No. 5, pp 423-433, May 1992..

[3] Gregg G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", IEEE Trans. Software Eng., vol. 22, no. 8, pp. 29-551, Aug. 1996.

[4] K.F. Fischer, F. Raji, and A. Ch ruscicki, "A Methodology for Retesting Modified Software", Proc. National Telecomm. Conf. B-6-3, pp. 1-6, Nov. 1981.

[5] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs", Proc. 20th ACM Symp. Principles of Programming Languages, Jan. 1993.

[6] M. Hutchins, H. Foster, T. Goradi a, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Control flow-Based Test Adequacy Criteria", Proc. 16th Int'l Conf. Software Eng., pp. 191- 2000, May 1994

[7] T.J. Ostrand and M.J. Balcer, "The Category-Pa rtition Method for Specifying and Generating Functional Tests", Comm. ACM, vol. 31, no. 6, June 1988.

[8] T.J. Ostrand and E.J. Weyuke r, "Using Dataflow Analysis for Regression Testing", Proc. Sixth Ann. Pacific Northwest Software Quality Conf., pp. 233-247, Sept. 1988.

[9] D. Rosenblum and G. Rothermel, "An Empirical Comparison of Regression Test Selection Techniques", Proc. Int'l Workshop Empirical Studies of Software Maintenance, pp. 89-94, Oct. 1997. Spector, A. Z. 1989.

[10] S.S. Yau and Z. Kishimoto, "A Method for Revalidating Modified Programs in the Maintenance Phase", Proc. 11th Ann. Int'l Computer Software and Applications Conf. (COMPSAC '87), pp. 272-277.

[11] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An empirical investigation of program spectra. SIGPLAN Notes 33, 7 (July 1998), 83-90.

[12] G. Ammons. Ball and .J. R. Laru s. Exploiting hardware performance counters with flow and context sensitive profiling. ACM Sigplan Notes, 32(5):85-~96, June 1997.

[13] T. Reps, T. Ball, M. Das, and .J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. ACM Software Engineering Notes, 22(6):432-439, Nov. 1997.

[14] W. E. Wong and V. Debroy, "A survey on software fault localization," Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, November 2009.

[15] Tiwari, S.; Mishra, K.K.; Kumar, A.; Misra, A.K.; , "Spectrum-Based Fault Localization in Regression Testing," *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on* , vol., no., pp.191-195, 11-13 April 2011 doi: 10.1109/ITNG.2011.40.

[16] M.-C. Gaudel, Testing can be fo rmal, too, Proceedings of the Sixth International Joint CAAP/FASE Conf erence on Theory and Practice of Software Development (TAPSOFT'95), Lecture Notes in Computer Science, vol. 915, Springer, Berlin, 1995, pp. 82–96.

[17] W.E. Howden, Reliability of the path analysis testing strategy, IEEE Transactions on Software Engineering SE-2 (3) (1976) 208–215.

[18] E.J. Weyuker, On testing non-testable programs, The Computer Journal 25 (4) (1982) 465–470.

[19] Gregg Rothermel, Mary Jean Harrold, "A safe, efficient regression test selection technique", ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 6 Issue 2, April „1997, Pages 173 – 210.

[20] Wong, J. R. Horgon, A. P. Mathur, and Pasquini, Test set size minimization and fault detection effectiveness: a case study in a space application", in proceedings of the IEEE Computer Society‟s International Computer Software and Applications Conference (COMPSAC‟97), pp. 522-528, Washington, DC, USA, August 1997.

[21] Anjaneyulu Pasala and et al.," Selection of Regression Test Suite to Validate Software Applications upon Deployment of Upgrades", IEEE 19th Australian Conference on Software Engineering, November, 2008.

[22] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of the Relationship between Spectra Differences and Regression Faults," Journal of Software Testing, Verification and Reliability, 10(3):171-194, September 2000.

[23] Yoo, S., & Harman, M. (2010). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*. doi:10.1002/stvr.430.

[24] T. Ball and J.R. Larus. Efficient Path Profiling in Proc. Of Micro 96, Pages 46-57, Dec 1996.

[25] Xie, T., Notkin, D.: Checking inside the black box: Regression testing by comparing value spectra. IEEE Transactions on Software Engineering 31(10), 869–883 (2005)

[26] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. InProceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 195–205, New York, 2004. ACM Press.

[27] Zhenyu Zhang, W.K. Chan, T.H. Tse, Y.T. Yu, Peifeng Hu, Non-parametric statistical fault localization, Journal of Systems and Software, Volume 84, Issue 6, June 2011, Pages 885-905, ISSN 0164-1212, 10.1016/j.jss.2010.12.048.

[28] Sample programs are taken for experiments from *Software Infrastructures Repository* (SIR) http://sir.unl.edu/content/sir.php.

[29] A. Orso and T. Xie. BERT: BEhavioral Regression Testing. In Proc. WODA, pages 36–42, 2008.