Checking Inside the Black Box: Regression Testing by Comparing Value Spectra

Tao Xie, *Member*, *IEEE*, and David Notkin, *Senior Member*, *IEEE*

Abstract—Comparing behaviors of program versions has become an important task in software maintenance and regression testing. Black-box program outputs have been used to characterize program behaviors and they are compared over program versions in traditional regression testing. Program spectra have recently been proposed to characterize a program's behavior inside the black box. Comparing program spectra of program versions offers insights into the internal behavioral differences between versions. In this paper, we present a new class of program spectra, value spectra, that enriches the existing program spectra family. We compare the value spectra of a program's old version and new version to detect internal behavioral deviations in the new version. We use a deviation-propagation call tree to present the deviation details. Based on the deviation-propagation call tree, we propose two heuristics to locate deviation roots, which are program locations that trigger the behavioral deviations. We also use path spectra (previously proposed program spectra) to approximate the program states in value spectra. We then similarly compare path spectra to detect behavioral deviations and locate deviation roots in the new version. We have conducted an experiment on eight C programs to evaluate our spectra-comparison approach. The results show that both value-spectra-comparison and path-spectra-comparison approaches can effectively expose program behavioral differences between program versions even when their program outputs are the same, and our value-spectra-comparison approach reports deviation roots with high accuracy for most programs.

Index Terms—Program spectra, regression testing, software testing, empirical studies, software maintenance.

1 Introduction

Regression testing retests a program after it is modified. In particular, regression testing compares the behavior of a new program version to the behavior of an old program version to assure that no regression faults are introduced. Traditional regression testing techniques use program outputs to characterize the behaviors of programs: When running the same test on two program versions produces different outputs (the old version's output is sometimes stored as the expected output for the test), behavior deviations are exposed. When these behavior deviations are unexpected, developers identify them as regression faults, and may proceed to debug and fix the exposed regression faults. When these behavior deviations are intended, for example, being caused by bug-fixing program changes, developers can be assured so and may update the expected outputs of the tests.

However, an introduced regression fault might not be easily exposed: Even if a program-state difference is caused immediately after the execution of a new faulty statement, the fault might not be propagated to the observable outputs because of the information loss or hiding effects. This phenomenon has been investigated by various fault models [20], [7], [29], [28]. Recently, a *program spectrum* has been proposed to characterize a program's behavior inside the black box of program execution [2], [24]. The name of *spectrum* comes from *path spectrum*, which is a distribution of paths derived from a run of the program. Some other program spectra, such as branch, data dependence, and execution trace spectra, have also been proposed in the literature [2], [13], [24].

In this paper, we propose a new class of program spectra called *value spectra*. The value spectra enrich the existing program spectra family [2], [13], [24] by capturing internal program states during a test execution. An internal program state is characterized by the values of the variables in scope. Characterizing behavior using values of variables is not a new idea. For example, Calder et al. [3] propose *value profiling* to track the values of variables during program execution. Our new approach differs from value profiling in two major aspects. Instead of tracking variable values at the instruction level, our approach tracks internal program states at each user-function entry and exit as the value spectra of a test execution. Instead of using the information for compiler optimization, our approach focuses on regression testing by comparing value spectra from two program versions.

When we compare the dynamic behavior of two program versions, a *deviation* is the difference between the value of a variable in a new program version and the corresponding one in an old version. We compare the value spectra from a program's old version and new version, and use the spectra differences to detect behavioral deviations in the new version. We use a deviation-propagation call tree to show the details of the deviations.

Some deviations caused by program changes might be intended such as by bug-fixing changes and some deviations might be unintended such as by introduced regression

- *T. Xie is with the Department of Computer Science, North Carolina State University, Box 8206, Raleigh, NC 27695-8206. E-mail: xie@csc.ncsu.edu.*
- *D. Notkin is with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350. E-mail: notkin@cs.washington.edu.*

Manuscript received 25 Mar. 2005; revised 9 June 2005; accepted 9 Sept. 2005; published online 3 Nov. 2005.
Recommended for acceptance by Harman, Korel, and Linos.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0079-0305.

faults. To help developers determine if the deviations are intended, it is important to present to developers the correlations between deviations and program changes. A *deviation root* is a program location in the new program version that triggers specific behavioral deviations. A deviation root is among a set of program locations that are changed between program versions. We propose two heuristics to locate deviation roots based on the deviation-propagation call tree. Identifying the deviation roots for deviations can help to understand the reasons for the deviations and determine whether the deviations are regression-fault symptoms or just expected. Identified deviation roots can be additionally used to locate regression faults if there are any.

A program state at a specific execution point can be approximated by using the path that the program execution traverses from the beginning of the program to the execution point. Then, in value spectra, we can replace the state representation using variable values with the state representation using traversed paths. The resulting path spectra correspond to complete-path spectra proposed in previous work [2], [13], [24]. We can similarly compare path spectra for exposing behavioral deviations and locating deviation roots.

This paper makes the following main contributions:

- We propose a new class of program spectra, called *value spectra*, to enrich the existing program spectra family [2], [13], [24] and extend the value profiling [3] used in compiler optimization. We present three variants of value spectra.
- We compare the value spectra from a program's old version and new version to detect behavioral deviations in the new version. We use a deviation-propagation call tree to show the details of the deviations.
- We propose two heuristics to locate deviation roots based on the deviation-propagation call tree.
- We extend previously proposed path spectra [2], [13], [24] to approximate value spectra and compare path spectra for detecting behavioral deviations and locating deviation roots.
- We conducted an experiment on eight C programs to evaluate our new approach. The experimental results show that our spectra-comparison approach can effectively report internal behavioral differences between program versions even when their program outputs are the same. Our deviation-root localization based on value-spectra comparison reports deviation roots with high accuracy for most programs.

The next section presents background information on regression testing and program spectra. Section 3 proposes value spectra. Section 4 describes how we exploit the differences between value spectra of the same test on two program versions. Section 5 proposes an approximation of value spectra by extending previously proposed path spectra [2], [13], [24]. Section 6 describes how we compare path spectra of the same test on two versions. Section 7 describes the experiment that we conducted to evaluate our spectra-comparison approach. Section 8 discusses related work and, then, Section 9 concludes.

2 Background

2.1 Regression Testing

Regression testing validates a modified program by retesting it. Regression testing is used to ensure that no new errors are introduced to a previously tested program when the program is modified. Because it is often expensive to rerun all tests after program modifications, one major research effort in regression testing is to reduce the cost of regression testing without sacrificing the benefit or sacrificing as little benefit as possible. For example, when some parts of a program are changed, regression test selection techniques [4], [25], [12] select a subset of the existing tests to retest the new version of the program. A *safe* regression test selection technique [25] ensures that the selected subset of tests contain all the tests that execute the code that was modified from the program's old version to new version. Sometimes the available resource might not even allow rerunning the subset of regression tests selected by regression test selection techniques. Recently, regression test prioritization techniques [33], [27], [8] have been proposed to order regression tests such that their execution provides benefits such as earlier detection of faults.

Regression-test quality is not always sufficient in exhibiting output differences caused by newly introduced errors in a program. Some previous test-generation approaches generate new tests to exhibit behavior deviations caused by program changes. For example, DeMillo and Offutt [7] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during mutation testing [6], [14]). Korel and Al-Yami [17] created driver code that compares the outputs of two program versions and then leveraged the existing white-box test-generation approaches to generate tests for which the two versions will produce different outputs. However, this type of test-generation problem is rather challenging and it is in fact an undecidable problem. The research in this paper tries to tackle the problem by exploiting the existing regression tests and checking more-detailed program behavior exercised inside the program.

2.2 Program Spectra

A program spectrum has been proposed to characterize a program's behavior [2], [24]. Although a program spectrum may characterize a program's behavior statically, a program spectrum is usually used in characterizing dynamic behavior exhibited by the execution of a test or multiple tests. One of the earliest proposed program spectra are *path spectra* [2], [24], [13]. Path spectra are represented by the executed paths in a program. There are variants of path spectra depending on whether to use the complete paths [13] or partial paths (loop-free intraprocedural paths) [2], [24], as well as whether to track the frequency of path occurrences. Harrold et al. [13] later proposed several types of program spectra (such as branch spectra, data-dependence spectra, execution-trace spectra, and output spectra) to investigate their potential applications in regression testing. *Branch spectra* consist of the set of conditional branches exercised by program execution. *Data-dependence spectra* consist of the set of definition-use pairs exercised by

program execution. *Execution-trace spectrum* consists of the sequence of program statements exercised by program execution. These program spectra as well as path spectra are defined by using the structural entities exercised by program execution. We refer to these types of program spectra as *syntactic spectra*. Harrold et al. [13] also proposed *output spectra*, which consist of the outputs produced by program execution. Ernst [9] developed the Daikon tool to detect likely program invariants from program execution and these dynamically detected invariants can also be considered as program spectra. Both output spectra and program invariants are defined by using program states (variable values) directly or indirectly. We refer these types of program spectra as *semantic spectra*.

Harrold et al. [13] define subsumption relationships among program spectra. Spectra type `S1` *subsumes* spectra type `S2` iff whenever the `S2` spectra for program `P`, version `P'`, and input `i` differ, the `S1` spectra for `P`, `P'`, and `i` differ. Spectra type `S1` *strictly subsumes* spectra type `S2` if `S1` subsumes `S2` and for some program `P`, version `P'`, and `i`, the `S1` spectra differ but the `S2` spectra do not. Spectra types `S1` and `S2` are *incomparable* if neither `S1` strictly subsumes `S2` nor `S2` strictly subsumes `S1`. In this work, we additionally define equivalence relationships among program spectra. Spectra types `S1` and `S2` are *equivalent* if `S1` subsumes `S2` and `S2` subsumes `S1`.

Harrold et al. [13] show that execution-trace spectra strictly subsume any other spectra. Path spectra strictly subsume branch spectra. Data-dependence spectra are incomparable to path spectra, branch spectra, or output spectra. Output spectra are incomparable to path spectra, branch spectra, or data-dependence spectra. In general, syntactic spectra except for execution-trace spectra are incomparable to semantic spectra. In addition, program invariants are incomparable to output spectra.

```
#include <stdio.h>
1 int max(int a, int b) {
2     if (a >= b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
8 int main(int argc, char *argv[]) {
9     int i, j;
10    if (argc != 3) {
11       printf("Wrong arguments!");
12       return 1;
13    }
14    i = atoi(argv[1]);
15    j = atoi(argv[2]);
16    if (max(i,j) >= 0){
17        if (max(i, j) == 0){
18            printf("0");
19        } else {
20            printf("1");
21        }
22    } else {
23        printf("-1");
24    }
25    return 0;
26 }
```

Fig. 1. A sample C program.

3 Value Spectra

This section introduces a new type of semantic spectra, value spectra, which are used to characterize program behavior. We first describe internal program state transitions in the granularity of user functions. Based on the internal program state transitions, we next define three variants of value spectra.

3.1 Internal Program State Transitions

The execution of a program can be considered as a sequence of internal program states. Each internal program state comprises the program's in-scope variables and their values at a particular execution point. Each program execution unit (in the granularity of statement, block, code fragment, function, or component) receives an internal program state and then produces a new one. The program execution points can be the entry and exit of a user-function execution when the program execution units are those code fragments separated by user-function call sites. Program output statements (usually output of I/O operations) can appear within any of those program execution units. Since it is relatively expensive in practice to capture all internal program states between the executions of program statements, we focus on internal program states in the granularity of user functions, instead of statements.

A *function-entry state* S^{entry} is an internal program state at the entry of a function execution. S^{entry} comprises the function's argument values and global variable values. A *function-exit state* S^{exit} is an internal program state at the exit of a function execution. S^{exit} comprises the function return value, updated argument values, and global variable values. Note that S^{exit} does not consider local variable values. If any of the preceding variables at the function entry or exit is of a pointer type, the S^{entry} or S^{exit} additionally comprises the variable values that are directly or indirectly reachable from the pointer-type variable. A *function execution* $\langle S^{entry}, S^{exit} \rangle$ is a pair of a function call's function-entry state S^{entry} and function-exit state S^{exit}.

To illustrate value spectra, we use a sample C program shown in Fig. 1. This program receives two integers as command-line arguments. The program outputs `-1` if the maximum of two integers is less than `0`, outputs `0` if the maximum of them is equal to `0`, and outputs `1` if the maximum of them is greater than `0`. When the program does not receive exactly two command-line arguments, it outputs an error message.

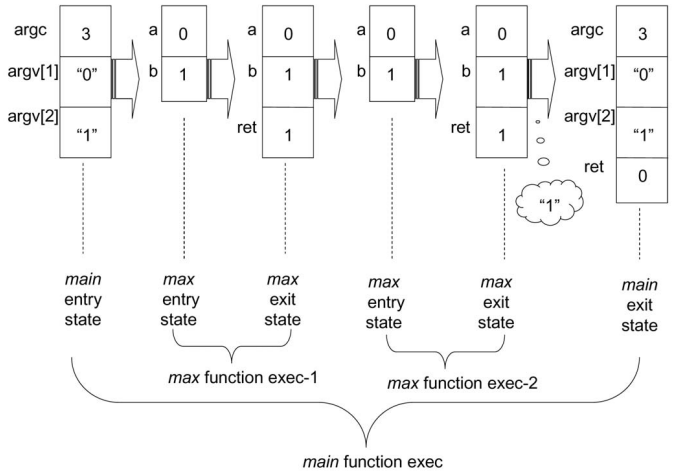

Fig. 2. Internal program state transitions of the sample C program execution with input "0 1".

Fig. 2 shows the internal program state transitions of the sample program with the command line arguments of "0 1". In the program execution, the main function calls the max function twice with the same arguments, and then outputs "1" as is shown inside the cloud in Fig. 2.

3.2 Value Spectra Types

We propose a new class of semantic spectra, *value spectra*, based on exercised internal program states. Value spectra track the variable values in internal program states, which are exercised as a program executes.

We propose three new variants of value spectra:

- *User-function value hit spectra* (in short as *value hit spectra*). Value hit spectra indicate whether a user-function execution is exercised.
- *User-function value count spectra* (in short as *value count spectra*). Value count spectra indicate the number of times that a user-function execution is exercised.
- *User-function value trace spectra* (in short as *value trace spectra*). Value trace spectra record the sequence of the user-function executions traversed as a program executes.

Table 1 shows different value spectra and output spectra for the sample C program execution with input "0 1". We represent a user-function execution using the following form:

`funcname(`*`entry`*`(argvals),`*`exit`*`(argvals,ret))`

where `funcname` represents the function name, `argvals` after *`entry`* represents the argument values and global variable values at the function entry, `argvals` after *`exit`* represents the updated argument values and global variable values at the function exit, and `ret` represents the return value of the function. Function executions in value hit spectra or value count spectra do not preserve order, while value trace spectra do preserve order. In value count spectra, a count marker of "* num" is appended to the end of each function execution to show that the function execution is exercised num times. Note that, if we change the second max function call from `max(i,j)` to `max(j,i)`, we will have two distinct entities for max in the value hit and value count spectra. It is because these two function executions will become distinct with different function-entry or function-exit states. In value trace spectra, "∨" markers are inserted in the function-execution sequence to indicate function execution returns [23]. The value trace spectra for the sample program shows that main calls max twice. Without these markers, the same function-execution sequence would result from main calling max and max calling max.

The value trace spectra strictly subsume the value count spectra, and the value count spectra strictly subsume the value hit spectra. The output spectra are incomparable with any of the three value spectra since the program's output statements inside a particular user function body might output some constants or variable values that are not captured in that user function's entry or exit state. For example, when we shuffle those printf statements in the main function body, the program still has the same value spectra but different output spectra. On the other hand, the executions with different value spectra might have the same output spectra. However, when those function bodies containing output statements are not modified in version P', the value trace spectra strictly subsumes the output spectra. In addition, if we also collect the entry and exit states of system output functions in the value trace spectra, the value trace spectra strictly subsume the output spectra.

Value trace spectra strictly subsume dynamically detected invariants because Ernst's Daikon tool [9] generalizes invariants from variable values that define value trace spectra. Because Daikon infers invariants for each function separately and the order among function executions does not affect the inference results, value count spectra also strictly subsume dynamically detected invariants. However, value

TABLE 1
Value Spectra for the Sample Program with Input "0 1"

spectra	profiled entities
value hit	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0)), max(`*`entry`*`(0,1),`*`exit`*`(0,1,1))`
value count	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0))*1, max(`*`entry`*`(0,1),`*`exit`*`(0,1,1))*2`
value trace	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0)), max(`*`entry`*`(0,1),`*`exit`*`(0,1,1)), ∨, max(`*`entry`*`(0,1),`*`exit`*`(0,1,1)), ∨, ∨`
output	"1"

hit spectra are not comparable to dynamically detected invariants because the number of data samples can affect Daikon's inference results [9]. For example, after we eliminate the second `max` method call by caching the return value of the first `max` method call, we will have the same value count spectra but Daikon might infer fewer invariants for `max` when running the two program versions with input `"0 1"` because too few data samples exhibit some originally inferred invariants.

Execution-trace spectra strictly subsume any other program spectra, including the three value spectra. Other syntactic spectra, such as branch, path, and data-dependence spectra are incomparable with any of the three value spectra. For example, when we change the statement of `i = atoi(argv[1])` to `i = atoi(argv[1]) + 1`, we will have the same traditional syntactic spectra, but different value spectra with input `"0 1"` running on the two program versions. On the other hand, when we move the statement of `printf("1")` from within the inner `else` branch to after the inner `else` branch, and add a redundant statement `i = i + 1` after the `printf("1")` statement, we will have different traditional syntactic spectra, but the same value spectra with input `"0 1"` running on the two program versions.

4 Value Spectra Differences

This section presents how we exploit the differences between value spectra of the same test on two program versions. We first describe how we compare value spectra. We then describe the deviation propagations exhibited by spectra differences. We finally present two heuristics to locate deviation roots based on deviation propagation.

4.1 Spectra Comparison

In this paper, we primarily focus on comparing value spectra from a program's old version and new version when we run the same test on these versions. We need to compare function executions from two program versions when comparing the value spectra from these versions. We can reduce the comparison of two function executions to the comparison of the function names, signatures, and the corresponding variable values in the function-entry and function-exit states from these two function executions. We next formally represent and compare function executions.

To represent a program state, we adapt Zimmermann and Zeller's formal notation for representing a program state as a rooted memory graph [36], where a vertex represents a value in memory and an edge between two vertices represents a reference between the values. Formally, let $G = (V, E, root)$ be a memory graph including a set V of vertices, a set E of edges, and a dedicated vertex $root$. Each vertex $v \in V$ is a triple $\langle val, tp, addr \rangle$, which represents a value val of type tp at memory address $addr$. Each edge $e \in E$ is also a triple $\langle v_1, v_2, op \rangle$, where $v_1, v_2 \in V$ are the starting and ending vertices of the edge and op is the name for the edge, being the variable name of the reference associated with the edge. A rooted memory graph for a function-entry state has a dedicated vertex $root \in V$ that references the function's arguments and global variables. A rooted memory graph for a function-entry state has a dedicated vertex $root \in V$ that references the function's return, arguments, and global variables. We perform a linearization algorithm [34] to linearize the rooted memory graph for a function-entry or function-exit state into a sequence, which is the representation of the state. The algorithm traverses the rooted memory graph starting from the root in the depth-first manner. For each first-time encountered vertex $v = \langle val, tp, addr \rangle$, we put into the sequence the name of the vertex (obtained by using the names on the path from the $root$ vertex to v[1]) and the value val. If an encountered vertex has been visited before, instead of putting val into the sequence, we put into the sequence the name of that vertex during the first-time visit. In the end of the graph traversal, the resulting sequence is the state representation.

Two states S_1 and S_2 are *equivalent* represented as $S_1 \equiv S_2$ if and only if their state representations are the same; otherwise are *nonequivalent*, represented as $S_1 \not\equiv S_2$. Two function executions $f_1 : \langle S_1^{entry}, S_1^{exit} \rangle$ and $f_2 : \langle S_2^{entry}, S_2^{exit} \rangle$ are *equivalent* if and only if they have the same function name and signature, $S_1^{entry} \equiv S_2^{entry}$, and $S_1^{exit} \equiv S_2^{exit}$. The comparison of value count spectra additionally considers the number of times that equivalent function executions are exercised. Given a function execution in the new version, the compared function execution in the old version is the one that has the same function name, signature, and function-entry state. If we cannot find such a function execution in the old version, the compared function execution is an *empty function execution*. An empty function execution has a different function name, function signature, function-entry state, or function-exit state from any other regular function executions.

The comparison of value trace spectra further considers the calling context and sequence order in which function executions are exercised. If we want to determine whether two value trace spectra are the same, we can compare the concatenated function-execution sequences of two value traces. If we want to determine the detailed function-execution differences between two value trace spectra, we can use the constructed dynamic call tree and the GNU Diffutils [11] to compare the function-execution traces of two value trace spectra. After the comparison, when a function execution f is present in Version `a` but absent in Version `b`, we can consider that an empty function execution in Version `b` is compared with f in Version `a`.

4.2 Deviation Propagation

Assume $f_{new} : \langle S_{new}^{entry}, S_{new}^{exit} \rangle$ is a function execution in a program's new version and $f_{old} : \langle S_{old}^{entry}, S_{old}^{exit} \rangle$ is its compared function execution in the program's old version. If f_{new} and f_{old} are equivalent, then f_{new} is a *nondeviated function execution*. If f_{new} and f_{old} are not equivalent, then f_{new} is a *deviated function execution*. We have categorized a deviated function execution into one of the following two types:

1. The path is constructed by using the sequence of vertices in the stack maintained during the depth-first search.

```
(The execution of the 58th test)
O main
 |__O initialize
 |__O alt_sep_test
     |__O Non_Crossing_Biased_Climb
     |  |__O Inhibit_Biased_Climb
     |  |__O Own_Above_Threat
     |__O Non_Crossing_Biased_Descend
     |  |__O Inhibit_Biased_Climb
     |  |__O Own_Below_Threat--------[dev follower]
     |  |__O ALIM--------------------[dev follower]
     |__O Own_Above_Threat

(The execution of the 91st test)
O main
 |__O initialize
 |__O alt_sep_test--------------------[dev container]
    |__O Non_Crossing_Biased_Climb
    |  |__O Inhibit_Biased_Climb
    |  |__O Own_Above_Threat
    |  |__O ALIM
    |__O Own_Below_Threat
    |__O Non_Crossing_Biased_Descend-[dev container]
       |__O Inhibit_Biased_Climb
       |__O Own_Below_Threat
```

Fig. 3. Value-spectra-based deviation-propagation call trees of a new program version (the ninth faulty version) of the `tcas` program.

- *Deviation container*. f_{new} is a deviation container if $S^{entry}_{new} \equiv S^{entry}_{old}$ but $S^{exit}_{new} \not\equiv S^{exit}_{old}$. If a function execution is identified to be a deviation container, developers can know that a certain behavioral deviation occurs *inside* the function execution. Note that when there is a certain behavioral deviation inside a function execution, the function execution might not be observed to be a deviation container, since the behavioral deviation might not be propagated to the function exit.
- *Deviation follower*. f_{new} is a deviation follower if $S^{entry}_{new} \not\equiv S^{entry}_{old}$. If a function execution is identified to be a deviation follower, developers can know that a certain behavioral deviation occurs *before* the function execution. For value count spectra particularly, a function execution in a program's new version can be categorized as a deviation follower if its count is different from the count of the compared function execution from the old program version. We need to use a matching technique (similar as the one used in the value trace spectra comparison) to identify which particular function executions in one version are absent in the other version.

The details of value spectra differences can provide insights into deviation propagation in the execution of the new program version. To provide such details, we attach deviation information to a dynamic call tree, where a vertex represents a single function execution and an edge represents calls between function executions. From the trace collected during a test execution, we first construct a dynamic call tree and then annotate the call tree with deviation information to form a deviation-propagation call tree. Fig. 3 shows the deviation-propagation call trees of two test executions on a new (faulty) version of the `tcas` program. The `tcas` program, its faulty versions, and test suite are contained in a set of `siemens` programs [15], which are used in the experiment described in Section 7. In the call trees, each node (shown as `O`) is associated with a function execution, and parent node calls its children nodes. For brevity, each node is marked with only the corresponding function name. The execution order among function executions is from the top to the bottom, with the earliest one at the top. If there is any deviated function execution, its deviation type is marked in the end of the function name.

Usually, behavioral deviations are originated from certain program locations that are changed in the new

program version. These program locations are called *deviation roots*. The function that contains a deviation root is called *deviation-root container*. In the new version of the `tcas` program, a relational operator > in the old version is changed to >=. The function that contains this changed line is `Non_Crossing_Biased_Descend`.

Some variable values at later points after a deviation-root execution might differ from the ones in the old program version because of the propagation of the deviations at the deviation root. The deviations at the function exit of the deviation-root container might cause the deviation-root container to be observed as a deviation container. Note that some callers of the deviation-root container might also be observed as deviation containers. For example, in the lower call tree of Fig. 3, the deviation-root container `Non_Crossing_Biased_Descend` is observed as a deviation container and its caller `alt_sep_test` is also observed as a deviation container.

Sometimes deviations after a deviation-root execution might not be propagated to the exit of the deviation-root container, but the deviations might be propagated to the entries of some callees of the deviation-root container, causing these callees to be observed as deviation followers. For example, in the upper call tree of Fig. 3, the deviation-root container's callees `Own_Below_Threat` and `ALIM` are observed as deviation followers.

4.3 Deviation-Root Localization

In the previous section, we have discussed how deviations are propagated given a known deviation root. This section explores the reverse direction: locating deviation roots by observing value spectra differences. This task is called *deviation-root localization*. Deviation-root localization can help developers to better understand which program change(s) caused the observed deviations and then determine whether the deviations are expected.

Recall that given a function execution $f_{new} : \langle S^{entry}_{new}, S^{exit}_{new} \rangle$, if f_{new} is a deviation container, S^{entry}_{new} is not deviated but S^{exit}_{new} is deviated; if f_{new} is a deviation follower, S^{entry}_{new} has already been deviated; if f_{new} is a nondeviated function execution, neither S^{entry}_{new} nor S^{exit}_{new} is deviated. Deviation roots are likely to be within those statements executed within a deviation container or before a deviation follower. The following two heuristics are to narrow down the scope for deviation roots based on deviation propagation effects:

Heuristic 1. Assume `f` is a deviation follower and `g` is the caller of `f`. If 1) `g` is a deviation container or a nondeviated one, and 2) any function execution between `g`'s entry and the call site of `f` is a nondeviated one, deviation roots are likely to be among those statements executed between the `g`'s entry and the call site of `f`, excluding user-function-call statements. For example, in the upper call tree of Fig. 3, `Own_Below_Threat` is a deviation follower and its caller `Non_Crossing_Biased_Descend` is a nondeviated one. The `Inhibit_Biased_Climb` invoked immediately before the `Own_Below_Threat` is a nondeviated one. Then, we can accurately locate the deviation root to be among those statements executed between the `Non_Crossing_Biased_Descend`'s entry and the call site of `Own_Below_Threat`.

Heuristic 2. Assume `f` is a deviation container. If any of `f`'s callees is a nondeviated one, deviation roots are likely to be among those statements executed within `f`'s function body, excluding user-function-call statements. For example, in the lower call tree of Fig. 3, `Non_Crossing_Biased_Descend` is a deviation container and any of its callees is a nondeviated one. Then, we can accurately locate the deviation root to be among those statements executed within the `Non_Crossing_Biased_Descend`'s function body.

When multiple changes are made at different program locations in the new program version, there might be more than one deviation root that cause behavioral deviations. If a deviation root's deviation effect is not propagated to the execution of another deviation root, and each deviation root causes their own value spectra differences, our heuristics can locate both deviation roots at the same time.

5 Path Spectra

This section presents three types of path spectra (extended from previously proposed path spectra [2], [13], [24]) to approximate an internal program state at a specific execution point. The state representation uses the path that the program execution takes from the beginning of the program to the execution point. In particular, a function-entry state S^{entry} is then represented by the path that the program execution takes from the beginning of the program to the entry point of the function, represented as P^{entry}. A function-exit state S^{exit} is represented by the path that the program execution takes from the beginning of the program to the exit point of the function P^{exit}. We can show P^{exit} to be P^{entry} concatenated with p, which is the path that the program execution takes within the function execution (taking into account the taken paths within the function's callees). Recall that a function execution $\langle S^{entry}, S^{exit} \rangle$ is characterized by S^{entry} and S^{exit}. After we use the path representation to approximate S^{entry} and S^{exit}, we can then define *user-function path hit spectra* (in short as *path hit spectra*), *user-function path count spectra* (in short as *path count spectra*), and *user-function path trace spectra* (in short as *path trace spectra*) corresponding to value hit spectra, value count spectra, and value trace spectra defined in Section 3.2. Note that our definitions of path spectra are slightly different from the ones defined in previous work [2], [24]. In previous work, path spectra refer to spectra that track partial paths—the set of loop-free intraprocedural paths exercised by program execution. Our definition of path spectra is more closely related to complete-path spectra [13].

The following is the path representation for the execution of the sample C program in Fig. 1 with input `"0 1"`: `8E,10F,1E,2F,5R,16T,1E,2F,5R,17F,25R`. The path representation consists of a sequence of branches taken by the program execution. Each branch is denoted by the line number of the corresponding conditional in the source code followed by "T" (denoting the true branch), "F" (denoting the false branch), "E" (denoting the method entry), and "R" (denoting the method return).

TABLE 2
Path Spectra for the Sample Program with Input "0 1"

spectra	profiled entities
path hit	main(*entry*(),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R,17F,25R)), max(*entry*(8E,10F),*exit*(8E,10F,1E,2F,5R)), max(*entry*(8E,10F,1E,2F,5R,16T),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R))
path count	main(*entry*(),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R,17F,25R))*1, max(*entry*(8E,10F),*exit*(8E,10F,1E,2F,5R))*1, max(*entry*(8E,10F,1E,2F,5R,16T),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R))*1
path trace	main(*entry*(),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R,17F,25R))*1, max(*entry*(8E,10F),*exit*(8E,10F,1E,2F,5R)), V, max(*entry*(8E,10F,1E,2F,5R,16T),*exit*(8E,10F,1E,2F,5R,16T,1E,2F,5R)),V, V

Table 2 shows three types of path spectra for the sample C program execution with input "0 1". Any of the three path spectra and complete-path spectra [13] subsume one another; therefore, these three path spectra and complete-path spectra are equivalent. In the rest of the paper, we shall discuss only path hit spectra (path spectra in short when it is clear from the context). Execution-trace spectra strictly subsume path hit spectra. Path hit spectra strictly subsume branch spectra. Path hit spectra are incomparable to data-dependence spectra, output spectra, value spectra, or program invariants.

6 Path Spectra Differences

We can compare the path hit spectra from running the same test on a program's old version and new version by using techniques similar to the ones for value spectra (Section 4.1). Fig. 4 shows the path-spectra-based deviation-propagation call trees of two test executions on a new (faulty) version of the `tcas` program (the same test executions for the value-spectra-based deviation call trees shown in Fig. 3). During the execution of the 58th test, the executed changed line in the faulty version lies within the method body of `Non_Crossing_Biased_Descend`, being after the execution of `Inhibit_Biased_Climb` and before the execution of `Own_Below_Threat`. A (first-encountered) diverged branch in the faulty version is taken before `Own_Below_Threat` is executed. The function-entry states of `Own_Below_Threat` and its subsequent executed functions are deviated; therefore, these function executions are identified as deviation followers. The function-exit states of `Non_Crossing_Biased_Descend` (the method containing the diverged branch) and its callers are deviated; therefore, these function executions are identified as deviation containers. During the execution of the 91st test, the executed changed line in the faulty version also lies within the method body of `Non_Crossing_Biased_Descend`, being after the execution of `Inhibit_Biased_Climb` and before the execution of `Own_Below_Threat`. A diverged branch in the faulty version is also taken before `Own_Below_Threat` is executed; therefore, `Own_Below_Threat` is identified as a deviation follower. `Non_Crossing_Biased_Descend` and its callers are identified as deviation containers.

Based on two call trees in Fig. 4, we can use Heuristic 1 (shown in Section 4.3) to accurately locate their deviation roots. In both call trees, the `Own_Below_Threat` function executions marked with * are deviation followers (but the `Own_Below_Threat` in the lower tree was a nondeviated one in Fig. 3 when we compared value spectra), and their callers `Non_Crossing_Biased_Descend` are deviation containers (but the `Non_Crossing_Biased_Descend` in the upper tree was a nondeviated one in Fig. 3 when we compared value spectra). In each tree, the `Inhibit_Biased_Climb` invoked immediately before the `Own_Below_Threat` is a nondeviated one. Then, we can use Heuristic 1 to accurately locate the deviation root to be among those statements executed between the `Non_Crossing_Biased_Descend`'s entry and the call site of `Own_Below_Threat`.

By comparing the call trees in Fig. 4 with the ones in Fig. 3, we can see that some nondeviated function executions in Fig. 3 are overconservatively identified as deviation containers or followers in Fig. 4 because the path representation is less precise for representing program states than the representation used in value spectra.

When multiple changes are made at different program locations in the new program version and more than one deviation root cause behavioral deviations, path-spectra-based heuristics can usually detect only the first deviation root at its best performance because the function-entry states of the later executed deviation roots are conservatively considered deviated.

7 Experiment

This section presents the experiment that we conducted to evaluate our approach. We first describe the experiment's objective and measures as well as the experiment instrumentation. We then present and discuss the experimental results. We finally discuss analysis cost and threats to validity.

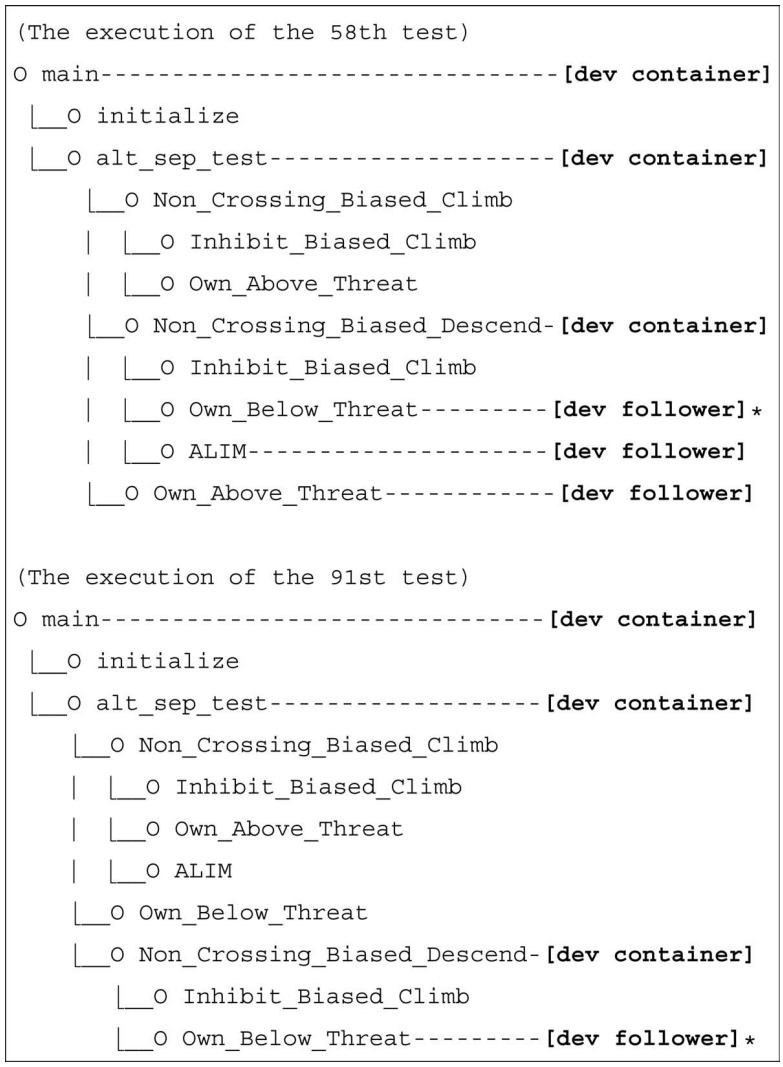

Fig. 4. Path-spectra-based deviation-propagation call trees of a new program version (the ninth faulty version) of the `tcas` program.

7.1 Objective and Measures

The objective of the experiment is to investigate the following questions:

1. How different are the three value spectra types, path spectra type, and output spectra type in terms of their deviation-exposing capability?
2. How accurately do the two deviation-root localization heuristics locate the deviation root from value spectra or path spectra?

Given spectra type S, program P, new version P', and the set CT of tests that cover the changed lines, let $DT(S, P, P', CT)$ be the set of tests each of which exhibits S spectra differences and $LT(S, P, P', CT)$ be the subset of $DT(S, P, P', CT)$ whose exhibited spectra differences can be applied with the two heuristics to accurately locate deviation roots. To answer Questions 1 and 2, we use the following two measures, respectively:

- *Deviation exposure ratio*. The deviation exposure ratio for spectra type S is the number of the tests in $DT(S, P, P', CT)$ divided by the number of the tests in CT, given by the equation:

$$\frac{|DT(S, P, P', CT)|}{|CT|}.$$

- *Deviation-root localization ratio*. The deviation-root localization ratio for spectra type S is the number of the tests in $LT(S, P, P', CT)$ divided by the number of the tests in $DT(S, P, P', CT)$, given by the equation: $\frac{|LT(S,P,P',CT)|}{|DT(S,P,P',CT)|}$.

Higher values of either measure indicate better results than lower values. In the experiment, we measure the deviation-root localization ratio in the function granularity for the convenience of measurement. That is, when the deviation-root localization locates the deviation-root containers (the functions that contain changed lines), we consider that the localization accurately locates the deviation root. For those changed lines that are in global data definition portion, we consider the deviation-root containers to be those functions that contain the executable code referencing the variables containing the changed data.

Sometimes the localization cannot accurately locate the deviation root; therefore, we additionally use the measure of *deviation-root localization distance* to show how difficult it is for developers to trace from an identified location to the

TABLE 3
Subject Programs Used in the Experiment

program	funcs	loc	tests	vers	vsgen (sec/test)	vscomp (sec/test)	psgen (sec/test)	pscomp (sec/test)	vssize (kb/test)	pssize (kb/test)
printtok	18	402	4130	7	0.76	0.18	0.14	0.04	6.51	0.92
printtok2	19	483	4115	10	0.48	0.08	0.19	0.05	1.72	1.19
replace	21	516	5542	32	0.49	0.08	0.18	0.02	2.1	0.85
schedule	18	299	2650	9	1.22	0.15	0.18	0.04	6.72	1.27
schedule2	16	297	2710	10	1.24	0.19	0.30	0.06	6.09	1.42
tcas	9	138	1608	41	0.35	0.04	0.03	0.02	0.36	0.23
totinfo	7	346	1052	23	0.51	0.04	0.13	0.02	1	0.4
space	135	6218	13585	18	1.46	0.23	0.28	0.07	28.43	4.03

deviation root. We compute the measure by counting the number of method entries and exits that the program execution encounters starting from the deviation root (the changed line in the faulty version) to the identified location (the diverged branch identified by path-spectra comparison). Lower values of the measure indicate better results than higher values.

7.2 Instrumentation

We built prototypes for the spectra-comparison approach to determine the practical utility. Our prototype for value-spectra comparison is based on Ernst et al.'s [9] Daikon Kvasir front end for C binaries [5]. Daikon is a system for dynamically detecting likely program invariants. It runs an instrumented program, collects and examines the values that the program computes, and detects patterns and relationships among those values. Based on a debugging and profiling tool called Valgrind [21], Daikon's Kvasir front end instruments C binaries for collecting data traces during program executions. By default, the Daikon front end instruments nested or recursive types (structs that have struct members) with the instrumentation depth of two and we set the instrumentation depth as three in the experiment. For example, given a pointer to the root of a tree structure, we collect the values of only those tree nodes that are within the tree depth of three. Our prototype for path-spectra comparison is based on the RECON instrumenter for C programs [32].

We have developed several tools (in Java) that compute and compare all three variants of value spectra, path hit spectra, and output spectra from the collected traces. In the experiment, we have implemented the deviation-root localization for only value hit spectra and path hit spectra.[2] Given two spectra, our tools report in textual form whether these two spectra are different. For value hit spectra and path hit spectra, our tools can display spectra differences in deviation-propagation call trees in plain text (as is shown in Figs. 3 and 4) and report deviation-root locations also in textual form.

We used eight C programs as subjects in the experiment. Researchers at Siemens Research created the first seven programs with faulty versions and a set of test cases [15]; these programs are popularly referred as the `siemens` programs (we used the programs, faulty versions, and test cases that were later modified by Rothermel and Harrold [26]). The researchers constructed the faulty versions by manually seeding faults that were as realistic as possible. Each faulty version differs from the original program by one to five lines of code. The researchers kept only the faults that were detected by at least three and at most 350 test cases in the test suite. The eighth program, called the `space` program, is a larger C program developed for the European Space Agency. The `space` program includes 9,564 lines of C code, among which 6,218 lines are executable. The `space` program is equipped with faulty versions, each of which contains a single fault that was exposed during the development of the `space` program. For the `space` program, Vokolos and Frankl [30] randomly created a test suite with 10,000 test cases and Rothermel et al. [27] augmented the test suite to include 3,585 additional test cases to achieve better structural coverage of the program.

Columns 1-5 of Table 3 show the program names, number of functions, lines of executable code, number of tests, number of faculty versions used in the experiment, respectively. Columns 6 and 8 show the average time (in seconds) of generating value spectra and path spectra, respectively, for one test. Columns 7 and 9 show the average time (in seconds) of comparing value spectra and path spectra, respectively, for one test. The last two columns show the average size (in kilobytes) of generated value spectra and path spectra, respectively, for one test.

We performed the experiment on a Linux machine with four Pentium IV 2.8 GHz processors. In the experiment, we used the original program as the old version and the faulty program as the new version. We used all the test cases in the test suite for each program.

7.3 Results

Figs. 5, 6, 7, and 8 use boxplots to present the experimental results. The box in a boxplot shows the median value as the

2. We have not implemented deviation-root localization for path count and path trace spectra because their localization results would be the same as the one of path hit spectra. We have not implemented deviation-root localization for value count or value trace spectra because their implementation requires the matching of traces from two versions, which is challenging by itself and beyond the scope of this research.

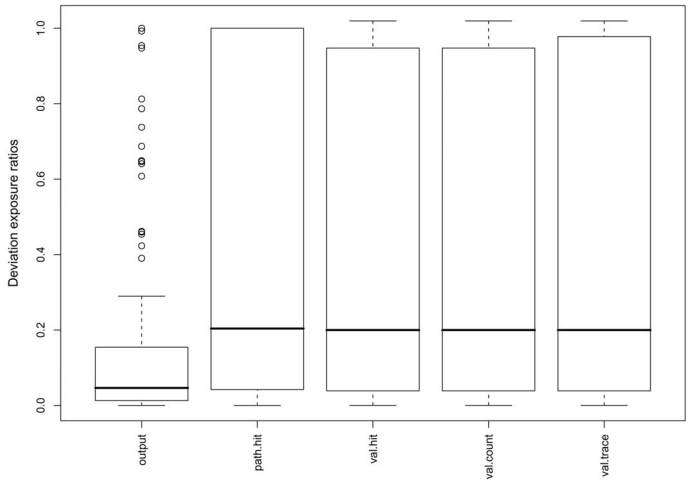

Fig. 5. Experimental results of deviation exposure ratios.

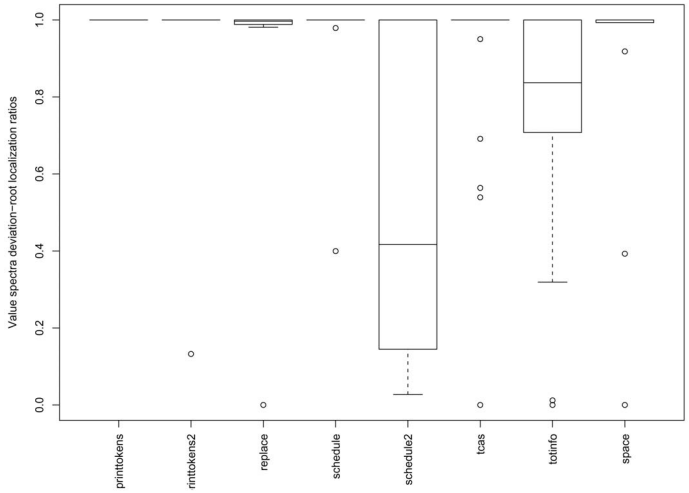

Fig. 6. Experimental results of deviation-root localization ratios for value hit spectra.

central line, and the first and third quartiles as the lower and upper edges of the box. The whiskers shown above and below the boxes technically represent the largest and smallest observations that are less than 1.5 box lengths from the end of the box. In practice, these observations are the lowest and highest values that are likely to be observed. Small circles beyond the whiskers are outliers, which are anomalous values in the data.

Fig. 5 shows the experimental results of deviation exposure ratios that are computed over all subjects. The vertical axis lists deviation exposure ratios and the horizontal axis lists five spectra types: output, path hit, value hit, value count, and value trace spectra. Figs. 6 and 7 show the experimental results of deviation-root localization ratios for value hit spectra and path hit spectra, respectively. The vertical axis lists deviation-root localization ratios and the horizontal axis lists subject names.

From Fig. 5, we observed that checking value spectra differences increases the deviation exposure ratio about a factor of three compared to checking program output differences. This indicates that a relatively large portion of deviations could not be propagated to program outputs. There are no significant differences of the deviation exposure ratios among the three value spectra, except that the third quartile of the value trace spectra is slightly higher than the one of the value hit or value count spectra. We found that there were three versions where value trace spectra have higher deviation exposure ratios than value hit and value count spectra. The faults in these three versions sometimes cause some deviation followers to be produced in value trace spectra, but these deviation followers are equivalent to some function executions produced by the old program version; therefore, although the value trace spectra are different, their value hit spectra or value count spectra are the same.

We observed that checking path spectra differences also increases the deviation exposure ratio compared to checking program output differences, and the increase is slightly more than checking value spectra differences. This indicates that some test executions change the execution paths, but do not exhibit deviations on program states at the entries or exits of user-defined functions.

In Fig. 6, the deviation-root localization ratios for value hit spectra are near 1.0 for all subjects except for the `schedule2` and `totinfo` programs; therefore, their boxes are collapsed to almost a straight line near the top of the figure. The results show that our heuristics for value hit spectra can accurately locate deviation roots for all subjects

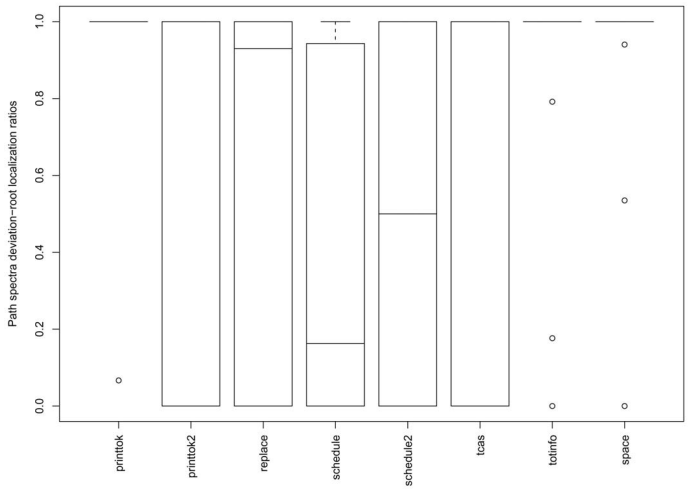

Fig. 7. Experimental results of deviation-root localization ratios for path hit spectra.

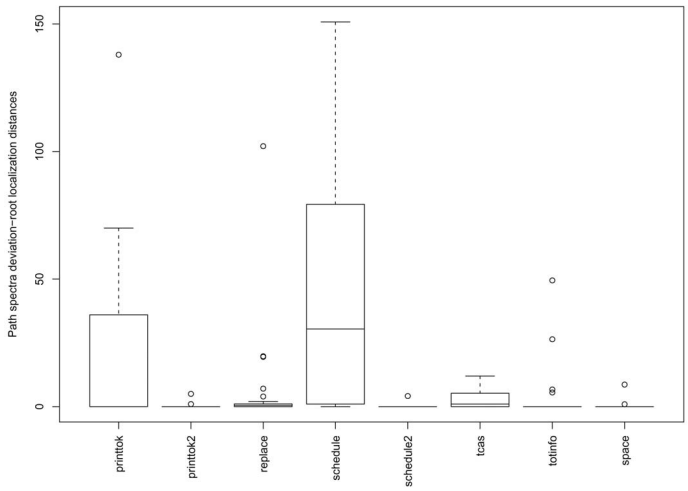

Fig. 8. Experimental results of deviation-root localization distances for path hit spectra.

except for the `schedule2` and `totinfo` programs. We inspected these two programs' traces carefully to find out the reasons. We found that the Daikon front end did not collect complete program state information in a key linked-list struct in `schedule2` using the instrumentation depth of three. In some of `schedule2`'s faulty versions, deviations occur on the key linked-list struct beyond the depth of three. Therefore, we could not detect the deviations at the exits of deviation roots. We expect that we could increase the deviation-root localization ratios after increasing the instrumentation depth. For `totinfo`, the Daikon front end did not collect sufficiently precise values for some variables with the `double` type; therefore, sometimes two double values are different during program execution. but their collected values in the string form are the same.

In Fig. 7, the deviation-root localization ratios for path hit spectra vary for different subjects and even different versions for the same subject. The results show that our heuristics for path hit spectra cannot locate deviation roots as accurately as the ones for value hit spectra. Only the first program (`printtok`) and the last two programs (`totinfo` and `space`) can achieve the ratio of near 1.0. We inspected the faulty versions of these three programs and found that most of their faults occur in the body of functions that contain few call sites after the faulty lines; during or after the execution of faulty lines, some different branches are taken before returning the function or invoking another function.

Fig. 8 shows the experimental results of deviation-root localization distances for path hit spectra. The vertical axis lists deviation-root localization distances and the horizontal axis lists subject names. To facilitate close observation of the data, we cut off `replace`'s two outliers, which are larger than 200. We observed that the localization distances vary across subjects. Two subjects have a rather high value (several dozens) for their localization distances. This indicates that it may be difficult for developers to trace from the identified location back to the deviation root for these subjects.

The experiment simulates the scenario of introducing regression faults into programs during program modifications. When programmers perform a modification that is not expected to change a program's semantic behavior, such as program refactoring [10], our spectra comparison approach can show the occurrences of unintended deviations and our deviation-root localization accurately locates the regression faults. Moreover, we can reverse the version order by treating the faulty version as the old version and the correct version as the new version. Then, we can

conduct a similar experiment on them. This simulates the scenario of fixing program bugs. Since our spectra comparison is symmetric, we expect to get the same experimental results. This shows that when programmers perform a bug-fixing modification, our approach can show them the occurrences of the intended deviations.

7.4 Analysis Cost

The time cost of our approach is primarily the time of generating spectra as well as comparing spectra (deviation-root localization is a part of spectra comparison). Columns 6 and 8 of Table 3 show the average time for generating value and path spectra, respectively, for one test. Columns 7 and 9 of Table 3 show the average time for comparing value and path spectra, respectively, for one test. The time cost of generating spectra is generally higher than that of comparing spectra and the time cost of the value-spectra approach is generally higher than that of the path-spectra approach. The elapsed time for generating or comparing two spectra of a test is less than one second, except for generating the value spectra of `schedule`, `schedule2`, and `space` (with the average time of 1.22, 1.24, and 1.46 seconds, respectively). The value spectra of these three programs are more expensive to generate because `schedule` and `schedule2` contain recursive data structures (which cause the size of a single function-entry or function-exit state to be relatively large) and `space` is a larger program (the size of its total function executions for a test is relatively large).

The space cost of our spectra-comparison approach is primarily the space for storing generated spectra. The space cost of storing value spectra is generally higher than that of storing path spectra. The space cost for the `space` program is much higher than that of other programs: The average space cost of storing its spectra is 28.43 kilobytes (KB) for one test's value spectra and 4.03 KB for one test's path spectra. The average space cost for the value spectra of `printtok`, `schedule`, or `schedule2` is above 6 KB and the cost for the remaining three programs ranges from 0.36 KB to 2.1 KB. Except for the `space` program, the space cost of all programs' path spectra is below 1.5 KB.

In general, larger programs require higher space and time costs. The time or space cost of our value-spectra-comparison approach can be approximately characterized as

$$VCost = O(|vars| \times |userfuncs| \times |testsuite|),$$

where $|vars|$ is the number of variables (including the pointer references reachable from the variables in scope) at the entry and exit of a user function, $|userfuncs|$ is the number of executed and instrumented user functions, and $|testsuite|$ is the size of the test suite.

The time or space cost of our path-spectra-comparison approach can be approximately characterized as

$$PCost = O(|branches| \times |userfuncs| \times |testsuite|),$$

where $|branches|$ is the number of executed branches within an instrumented user function.

We have incorporated the following two mechanisms in our tool implementation to reduce the analysis cost. First, our implementation postprocesses the data traces collected by the Daikon front end by filtering out (meaningless) uninitialized variables. Second, our implementation stores generated spectra files in a compressed form. To further reduce the analysis cost, we can reduce $|testsuite|$ by applying our approach on only those tests selected by regression test selection techniques [25]. In addition, we can also reduce $|userfuncs|$ by instrumenting only those modified functions and their (statically determined) up-to-n-level callers or those functions enclosed by identified firewalls [19], [31]. The reduced scope of instrumentation trades a global view of deviation propagation for efficiency. Specifically for path spectra, we can use partial paths [2], [24] instead of complete paths to reduce cost.

7.5 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, faults or program changes, and test cases are representative of true practice. The `siemens` programs are small and the `space` program is of medium size. Most of the faulty versions involve simple, one or two-line manually seeded faults. Moreover, the new versions in our experiment do not incorporate other fault-free changes since all the changes made on faulty versions deliberately introduce regression faults. These threats could be reduced by more experiments on wider types of subjects in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype, the Daikon front end, and the RECON instrumenter might cause such effects. To reduce these threats, we manually inspected the spectra differences on a dozen of traces for each program subject. One threat to construct validity is that our experiment makes use of the data traces collected during executions, hoping that these precisely capture the internal program states for each execution point.

8 Related Work

Reps et al. [24] compare path spectra (loop-free intraprocedural paths) from the executions of two tests on the same program to tackle the Year 2000 problem. Our approach compares value spectra or path spectra from the execution of the same test on two program versions to tackle the regression testing problem. Our path-spectra comparison approach adopts a slightly different definition of path spectra and we compare the effectiveness of path-spectra comparison with our value-spectra comparison. Our experimental results show that the deviation-root localization based on value spectra performs more accurately than the one based on path spectra.

Harrold et al. [13] empirically investigate the relationship between syntactic spectra differences and output spectra differences of two program versions in regression testing. The syntactic spectra that they investigate include branch, path, data dependence, and execution trace spectra. Their experimental results show that, when a test input causes program output differences between versions, the test input is likely to cause syntactic spectra differences. However, their results show that the reverse is not true. Our experimental results on path-spectra comparison confirm their observations and we intend to take advantage of this observation to expose more behavioral deviations by comparing program spectra.

Abramson et al. [1] develop the relative debugging technique that requires users to specify key data structures that must be equivalent at specific locations in two program versions. Then, a relative debugger automatically compares the data structures and reports any differences while both program versions are executed concurrently. Our approach does not require user-defined locations but compares states at the entries and exits of user functions. In principle, to save space cost, comparing value spectra can also be done online while running two program versions simultaneously instead of our current offline analysis on collected traces. However, one challenge in online comparison is to automatically and accurately correlate and synchronize corresponding program parts during the simultaneous execution of two program versions.

Jaramillo et al. [16] develop the comparison checking approach to compare the outputs and values computed by source level statements in the unoptimized and optimized versions of a source program. Their approach requires the optimizer writer to specify the mappings between the unoptimized and optimized versions in the optimization implementation. Their approach locates the earliest point the unoptimized and optimized programs differ during the comparison checking. Our approach operates at the granularity of user-function executions and uses two heuristics to locate deviation roots instead of using the earliest deviation points (coincidentally, the heuristics based on path spectra identify the earliest diverged branch to belong to deviation roots). Moreover, our approach does not require any extra user inputs and targets at testing general applications rather than optimizers.

Fault propagation has been investigated in the testing literature [20], [7], [29], [28]. These models primarily focus on the estimation and analysis of fault exposure probability with the goal of generating or selecting test cases that propagate the faults to outputs. Our approach focuses on regression testing and proactively exposes behavioral deviations by checking inside the black box instead of checking only black-box outputs. Our approach also offers an empirical way of studying fault propagation behavior complementing existing analytic approaches.

Weak mutation testing [14] requires that a test case causes a mutated version to compute a different value than the original version does, in contrast to the strong mutation testing [6] that requires mutated and original versions to produce different outputs. The difference between our approach and traditional output-checking approach is analogous to weak and strong mutation testing. Neither weak mutation testing nor our approach requires the faults to be propagated to outputs. Weak mutation testing observes the exposure of hypothesized and seeded faults by a test for estimating and measuring the test, whereas our approach observes the exposure of actual faults or behavioral deviation by a test for regression testing. Moreover, weak mutation testing is primarily a unit testing technique, but our approach can be either system or unit testing technique by checking internal behavior inside a system or unit.

Reese and Leveson [22] present the software deviation analysis technique to determine whether a software specification can behave well when there are deviations in data inputs from an imperfect environment. Their software deviation analysis detects behavioral changes between two identical programs given slightly different inputs, whereas our approach detects behavioral changes between two program versions given the same inputs. The deviations in our approach are rooted from some program locations that are changed between versions, rather than from program inputs.

A dynamic slice [18] of a program is the set of statements that actually affect the value of a selected variable at a specific location after the program is executed against a given test. Dynamic slicing uses runtime structural information in addition to some static information. A dynamic slice for a deviated variable may include multiple program changes, even if only one of them actually causes the deviation, whereas our deviation-root localization uses state information and may single out the responsible program change. Our approach reports deviation roots in the granularity of functions or code segments separated by call sites. The deviated variables at a function's entry or exit (produced by our approach) can be inputs to the dynamic slicing approach, which can report deviation roots in the statement level within a function.

9 Conclusion

After developers made changes on their program, they can rerun the program's regression tests to assure the changes take effect as intended: refactoring code to improve code quality, enhancing some functionality, fixing a bug in the code, etc. To help developers to gain a higher confidence on their changes, we have proposed a new approach that checks program behaviors inside the black box over program versions besides checking the black-box program outputs.

We have developed a new class of semantic spectra, called value spectra, to characterize program behaviors. We exploit value spectra differences between a program's old version and new version in regression testing. We use these value spectra differences to expose internal behavioral deviations inside the black box. We also investigate deviation propagation and develop two heuristics to locate deviation roots. If there are regression faults, our deviation-root localization additionally addresses the regression fault localization problem. We also approximate value spectra by extending previously proposed path spectra [2], [13], [24] and detect behaviorial deviations and locate deviation roots by similarly comparing path spectra. We have conducted an experiment on eight C program subjects. The experimental results show that both value-spectra-comparison and path-spectra-comparison approaches can effectively detect behavioral deviations even before deviations are (or even if they are not) propagated to outputs. The results also show that our deviation-root localization based on value spectra can accurately locate deviation roots for most subjects.

Acknowledgments

This is a revised and extended version of a paper that appeared at the 2004 International Conference on Software Maintenance (ICSM '04) [35]. The authors would like to thank Michael Ernst, the Daikon project members at MIT, Mary Jean Harrold, and Gregg Rothermel for helping with experiment instrumentation. We thank Miryung Kim, Andrew Peterson, Gregg Rothermel, Vibha Sazawal, and

the ICSM '04 and *IEEE Transactions on Software Engineering* anonymous reviewers for their valuable feedback on an earlier version of this paper. This work was supported in part by the US National Science Foundation under grant ITR 0086003 and through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

REFERENCES

[1] D. Abramson, I. Foster, J. Michalakes, and R. Socic, "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM,* vol. 39, no. 11, pp. 69-77, 1996.
[2] T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. 29th Int'l Symp. Microarchitecture,* pp. 46-57, 1996.
[3] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 259-269, 1997.
[4] Y.-F. Chen, D.S. Rosenblum, and K.-P. Vo, "TestTube: A System for Selective Regression Testing," *Proc. 16th Int'l Conf. Software Eng.,* pp. 211-220, 1994.
[5] Daikon invariant detector tool, 2005, http://pag.csail.mit.edu/daikon/download/.
[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34-41, Apr. 1978.
[7] R.A. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. Software Eng.,* vol. 17, no. 9, pp. 900-910, Sept. 1991.
[8] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 159-182, Feb. 2002.
[9] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 99-123, Feb. 2001.
[10] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.
[11] GNU, GNU Diffutils, http://www.gnu.org/software/diffutils/, 2002.
[12] T.L. Graves, M.J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Trans. Software Eng. Methodology,* vol. 10, no. 2, pp. 184-208, 2001.
[13] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of the Relationship between Spectra Differences and Regression Faults," *J. Software Testing, Verification and Reliability,* vol. 10, no. 3, pp. 171-194, 2000.
[14] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.,* vol. 8, no. 4, pp. 371-379, July 1982.
[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* pp. 191-200, 1994.
[16] C. Jaramillo, R. Gupta, and M.L. Soffa, "Debugging and Testing Optimizers through Comparison Checking," *Proc. Int'l Workshop Compiler Optimization Meets Compiler Verification,* Apr. 2002.
[17] B. Korel and A.M. Al-Yami, "Automated Regression Test Generation," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 143-152, 1998.
[18] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters,* vol. 29, no. 3, pp. 155-163, 1988.
[19] H.K.N. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," *Proc. Int'l Conf. Software Maintenance,* pp. 290-300, 1990.
[20] L.J. Morell, "A Theory of Fault-Based Testing," *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 844-857, Aug. 1990.
[21] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Proc. Third Workshop Runtime Verification,* July 2003.
[22] J.D. Reese and N.G. Leveson, "Software Deviation Analysis," *Proc. Int'l Conf. Software Eng.,* pp. 250-260, 1997.
[23] S.P. Reiss and M. Renieris, "Encoding Program Executions," *Proc. Int'l Conf. Software Eng.,* pp. 221-230, 2001.
[24] T. Reps, T. Ball, M. Das, and J. Larus, "The Use Of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *Proc. Sixth European Software Eng. Conf. (ESEC) and Seventh ACM SIGSOFT Int'l Symp. the Foundations of Software Eng.,* pp. 432-449, 1997.
[25] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Software Eng. Methodology,* vol. 6, no. 2, pp. 173-210, 1997.
[26] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proc. Int'l Conf. Software Maintenance,* pp. 34-43, 1998.
[27] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Oct. 2001.
[28] M.C. Thompson, D.J. Richardson, and L.A. Clarke, "An Information Flow Model of Fault Detection," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 182-192, 1993.
[29] J.M. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 717-727, July 1992.
[30] F.I. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenance,* pp. 44-53, 1998.
[31] L. White and H.K.N. Leung, "A Firewall Concept for Both Control-Flow and Data-Flow in Regression Integration Testing," *Proc. Int'l Conf. Software Maintenance,* pp. 262-271, 1992.
[32] N. Wilde The RECON Software Reconnaissance Tool, Feb. 2003, http://www.cs.uwf.edu/recon/.
[33] W.E. Wong, J.R. Horgan, S. London, and H.A. Bellcore, "A Study of Effective Regression Testing in Practice," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 264-274, 1997.
[34] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," *Proc. 19th Int'l Conf. Automated Software Eng.,* pp. 196-205, Sept. 2004.
[35] T. Xie and D. Notkin, "Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences," *Proc. Int'l Conf. Software Maintenance,* pp. 28-37, Sept. 2004.
[36] T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," *Dagstuhl Seminar on Software Visualization,* 2001.

Tao Xie received the PhD degree at the University of Washington in 2005. He received the BS degree from Fudan University in 1997 and the MS degree from Peking University in 2000. He is an assistant professor in the Department of Computer Science at North Carolina State University. His primary research interest is software engineering, with an emphasis on techniques and tools for improving software reliability and dependability. He is a member of the IEEE.

David Notkin received the ScB degree at Brown University in 1977 and the PhD degree at Carnegie Mellon University in 1984. He is the Bradley Professor and Chair of Computer Science and Engineering at the University of Washington. Before joining the faculty in 1984, Dr. Notkin received the US National Science Foundation Presidential Young Investigator Award in 1988; served as the program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering; served as program cochair of the 17th International Conference on Software Engineering; chaired the Steering committee of the International Conference on Software Engineering (1994-1996); served as charter associate editor of both the *ACM Transactions on Software Engineering and Methodology* and the *Journal of Programming Languages*; served as an associate editor of the *IEEE Transactions on Software Engineering*; was named as an ACM Fellow in 1998; served as the chair of ACM SIGSOFT (1997-2001); and received the University of Washington Distinguished Graduate Mentor Award in 2000. His research interests are in software engineering in general and in software evolution in particular. He is a senior member of the IEEE.