Checking Inside the Black Box: Regression Fault Exposure and Localization Based on Value Spectra Differences

Tao Xie David Notkin
Department of Computer Science & Engineering, University of Washington
{taoxie, notkin}@cs.washington.edu

Technical Report UW-CSE-02-12-04
December 2002

Abstract

We present a new fault exposure and localization approach intended to increase the effectiveness of regression testing. In particular, we extend traditional regression testing, which strongly focuses on black box comparisons, to compare internal program states. These value spectra differences *allow a more detailed comparison of executions of the new and old versions of a program. In particular, our approach checks inside the program black box to observe unit behaviors and further checks inside the unit black box to observe some internal variable values besides the ones propagated outside the unit. This approach exposes faults without requiring the faults to be propagated to the outputs of the system or unit. Two heuristics are proposed to locate regression faults based on a fault propagation model. An experiment is conducted to assess their effectiveness. The initial results show our value-spectra-comparison approach can increase the regression fault exposure probability effectively and identify the locations of regression faults accurately.*

1. Introduction

"From the value of testing perspective, information hiding reduces the ability for faults to propagate to an observable output and hence reduces the likelihood that faults will be revealed during testing" [15].

Traditional regression testing compares behaviors of a new version to the behaviors of an old version in the hope of exposing any introduced fault. When the outputs produced by two versions are different, regression faults are exposed. However, even if a variable value difference is caused immediately after a new faulty statement is executed, the fault might not be propagated to the observable outputs due to the information loss or hiding effects. Checking inside the black box has been used to expose faults complementing the traditional black box output checking approach. Runtime assertion or inferred invariant checking is used to validate that certain properties inside the black box are satisfied during program execution [3][6][9][18]. In the regression testing context, comparing structural program spectra inside the black box between versions, e.g. path, branch, or execution trace spectra, etc., has been investigated [10]. The experimental results from this work show that when the black box outputs between versions are different, the structural program spectra are likely also to be different, though unfortunately the reverse is not true. This leads to a hypothesis that structural program spectra might not sufficiently capture a program's stable behaviors, which are supposed to be kept the same across versions. For example, restructuring or refactoring can change a program's structural program spectra but still preserve the same meaning [7][8].

To check relatively stable behaviors inside the black box across versions, we propose a new type of program spectra called *value spectra*. Value spectra capture internal program states during test executions, which are expected to be more stable across versions. Value spectra differences are used to expose regression faults and assist fault localization. Mapping and filtering mechanisms are used to accommodate those changes that might cause value spectra differences that do not introduce regression faults. The next section discusses value spectra differences. Section 3 presents the applications of value spectra differences in regression fault exposure and localization. Section 4 describes the experiment that is conducted to assess our approach. Section 5 discusses related work. Finally, Section 6 concludes with a discussion.

2. Value Spectra Differences

2.1. Internal Program State Transition

The execution of a program can be considered as a sequence of internal program states [22] . Each internal program state comprises the program's 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 points of a user function execution when the program execution units are those code fragments separated by user function call sites. Program output statements (usually I/O output operations) can appear within any of those program execution units. Figure 2 shows the internal program state transition chain of a simple C sample in Figure 1 when the internal program states are captured in the entry and exit points of user functions.

In an internal program state, those variable values at the entry point of a user function execution comprise that function's argument values and global variable values, which are captured by a memory snapshot at the function entry point. Those variable values at the function exit point comprise function return values, parameter passing out values, and global variable values, which are visible outside the function. At the function exit point, although some argument values are not visible externally, they are still meaningful with respect to program validation since the post-condition specification of that function usually involves these values. Therefore, the values of all argument variables at the function exit point are also captured in the internal program state for that function exit point. The variable values in the internal program states at the entry and exit points of a function execution form an entry-exit variable value pair for that function execution, representing the input/output of that function execution for our use.

2.2. Value Spectra

A program spectrum characterizes a program's behavior [17]. Several classes of program spectra are proposed in literature, including path spectra, branch spectra, data-dependence spectra, execution trace spectra, and output spectra [4][10]. Except for output spectra, which record the outputs produced by a program as it executes, these spectra are based on execution structure. A new class of program spectra, value spectra, is proposed in this research. Value spectra track the variable values in internal program states, which are exercised as a program executes. Output spectra can be viewed as a special kind of value spectra, recording the system outputs produced by a program as it executes.

In this research, four distinct types of value spectra are proposed:

- **Program output value spectrum (POV).** A *program output value spectrum* records the output values produced by a program as it executes. It is the same as the earlier output spectrum [10].
- **Function value hit spectrum (FVH).** For each entry-exit variable value pair for a function execution, a *function value hit spectrum* indicates whether or not that pair is exercised.
- **Function value count spectrum (FVC).** For each entry-exit variable value pair for a function execution, a *function value count spectrum* indicates the number of times that pair is exercised.
- **Execution-trace value spectrum (ETV).** The *execution-trace value spectrum* records the sequence of the entry-exit variable value pairs for the functions traversed as a program executes.

```
/* sample.c - Sample C program to output 2
when 0 < max < 10, output 1 when max >= 10,
and output 0 when max <= 0, where max is the
maximum of two integers. If the program
receives fewer or more than two integers, it
outputs "Wrong arguments!" message. */

#include <stdio.h>

int max(int a, int b)
{
      if (a >= b) {
            return a;
      } else {
            return b;
      }
}

int main(int argc, char *argv[])
{
      int i, j;
      if (argc != 3) {
            printf("Wrong arguments!");
      }
      i = atoi(argv[1]);
      j = atoi(argv[2]);
      if (max(i,j) > 0){
            if (max(i, j) < 10){
                  printf("2");
            } else {
                  printf("1");
            }
      } else {
            printf("0");
      }
      return 0;
}
```

Figure 1. A sample C program

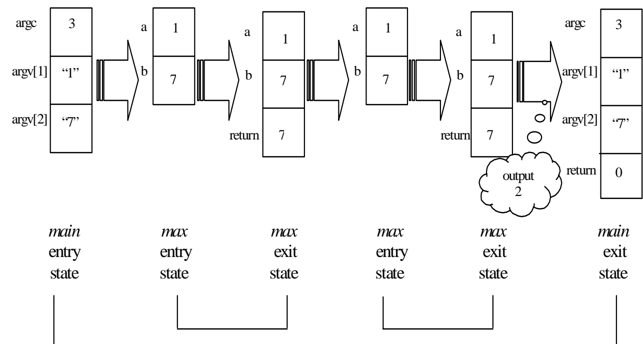

Figure 2. Internal program state transition chain of the sample C program execution with input "1 7"

FVH, FVC, and ETV reflect the internal program states as a program executes. Table 1 shows the value spectra for the sample C program execution with input "1 7". Profiled entities of value spectra are those entry-exit variable value pairs for user function executions. Profiled entities of FVH or FVC have no ordering among them but those of ETV do have. In the end of each FVC profiled entity, a count marker of "* *NUM*" is used to show that

profiled entity is exercised *NUM* times. Returning markers of "∨" are inserted in ETV profiled entity sequence to indicate a call hierarchy [16].

Table 1. Value spectra for the sample program with input "1 7"

Spectra	Profiled Entities
FVH	Smain3_"1"_"7"|3_"1"_"7"_0, Smax1_7|1_7_7
FVC	Smain3_"1"_"7"|3_"1"_"7"_0 * 1, Smax1_7|1_7_7 * 2
ETV	Smain3_"1"_"7"|3_"1"_"7"_0, Smax1_7|1_7_7, ∨, Smax1_7|1_7_7, ∨, ∨
POV	Output 2
Smain3_"1"_"7"| 3_"1"_"7"_0: main:entry(argc=3,argv[1]="1",argv[2]="7"), exit(argc=3, argv[1]="1",argv[2]="7",return=0) Smax1_7|1_7_7: max:entry(a=1,b=7), exit(a=1,b=7,return=7)	

Spectrum type *S1* subsumes spectrum type *S2* if and only if whenever the *S2* spectrum for program *P*, version *P'*, and input *i* differ, the *S1* spectrum for *P*, *P'*, and *i* differ. Spectrum type *S1* strictly subsumes spectrum type *S2* if *S1* subsumes *S2* and for some program *P*, version *P'*, and *i*, the *S1* spectrum differs but the *S2* spectrum does not. Spectrum types *S1* and *S2* are incomparable if neither *S1* strictly subsumes *S2* nor *S2* strictly subsumes *S1* [10].

ETV strictly subsumes FVC and FVC strictly subsumes FVH. POV is incomparable with any of ETV, FVC, and FVH, since the program output statements inside a particular function body might output the variable values that are not captured in that function's entry-exit variable value pair. For example, when those *printf* statements in *main* function body are shuffled, the program still has the same spectra of ETV, FVC, or FVH, but different POV spectra. On the other hand, the executions with different ETV, FVC, or FVH spectra might have the same POV spectra. However, when those function bodies containing program output statements are not modified in version *P'*, ETV strictly subsumes POV.

3. Applying Value Spectra Differences

3.1. Regression Fault Exposure

Among those value spectra that reflect internal program states, FVH are most stable during program evolution because FVH does not consider structural information or function execution counts, which are relatively volatile during program evolution. In our approach, FVH differences are used to expose regression faults. However, two kinds of FVH differences are handled specially to accommodate some common fault-free changes that cause these FVH differences. When there is a FVH entry-exit variable value pair in the old version but not in the new version, no regression faults are reported because of that, since it is common that a function execution in the old version might be inlined or removed in the new version. Moreover, when there is a FVH entry-exit variable value pair in the new version but not in the old version and the function for that value pair is newly added in the new version, no regression faults are reported. Therefore, only when there is a FVH entry-exit variable value pair in the new version but not in the old version and the function for that value pair exists in the old version, are regression faults reported. In the context of our approach, FVH differences between the old and the new versions refer to this case alone. Moreover, when some variables in a variable value pair are pointers, the actual data content eventually pointed by them are compared rather than the pointers themselves, since all pointers can have different values in alternative runs but still have the same semantics. Memory graph comparison can compare the entire structure among pointers in addition to the data pointed by them but might not be scalable [23].

3.2. Fault Propagation

The details of value spectra differences can show how internal state transitions in a later version deviate from the ones in an earlier version. These differences can provide insights into fault propagation in the execution of the later version. There are two primary reasons that faults are not found by techniques that check for differences in the unit or program outputs. The first reason is that immediately after the execution of faulty code, environmental values, such as variable values in scope or outputs to external I/O devices, might be the same as the ones when the correct code is executed. The second reason is that even when the environmental values are different than the correct ones immediately after the execution of the faulty code, which is called immediate value infection, this value deviation might not be propagated to observable unit or program outputs. Our approach may expose faults that are not found by traditional techniques due to the second reason.

Some variable values at certain points after faulty code execution might differ from the correct one due to the propagation of immediate value infection. This phenomenon is called propagated value infection. The variables with different values from the correct ones are called infected variables. The value infection in the entry or exit point of a function execution can cause FVH differences being observed in our approach.

Sometimes an immediate value infection might not be propagated to the exit point of a faulty function but might be propagated to the entry points of some callees of that faulty function. Sometimes an immediate value infection might be propagated to the exit point of that faulty

function, but the scopes of all the infected variables are limited in that function, not being visible outside that function. Sometimes an immediate value infection is propagated outside that faulty function to its call site and further to the entry point of some functions executed later. If a value infection is finally propagated to program outputs, checking the outputs can expose a regression fault.

3.3. Fault Localization Heuristics

Those locations of faults can be pinpointed based on details of FVH differences according to a fault propagation model. In particular, there is an enclosing relationship between two variable value pairs if the entry point of the former pair is executed before the entry point of the latter one but its exit point is executed after the exit point of the latter one. If a variable value pair encloses another one, the function invocation corresponding to the former pair is the ancestor caller of the one corresponding to the latter pair. The ordering among variable value pairs is sorted based on the execution time of their entry points with the earliest first. We have defined two types of FVH differences:

- **Entry-Same-eXit-Diff (ESXD).** In a variable value pair for a function execution in the new version, the entry point variable values are the same as those in a pair for the same function execution in the old version, but the exit point variable values are different. Notice that if a program crash site occurs within a function execution in the new version, there are no exit point variable values available for that function execution in the new version; thus the exit point variable values differ from those in the old version.
- **Entry-Diff (ED).** In a variable value pair for a function execution in the new version, the entry point variable values are different from those in any pair for the same function execution in the old version. As is discussed in Section 3.1, in the new version a variable value pair of a newly created function is filtered out, not being considered as FVH differences.

An ESXD's variable values at the entry point are not infected but its variable values at the exit point are infected. In contrast, an ED's variable values at the entry point have already been infected. Those variable value pairs that do not show FVH differences are usually not infected in either their entry or exit points. Our fault localization heuristics are described as follows:

- **Heuristic 1.** If the variable value pair for an ED's caller execution is not an ED and neither an ED nor an ESXD is present between the variable value pair for that ED's caller execution and that ED, fault locations are likely to be those statements executed between the ED's caller and that ED's call site.
- **Heuristic 2.** If an ESXD encloses neither an ED nor another ESXD, fault locations are likely to be those statements executed within the ESXD's function body.

The fault locations are likely to be within those statements executed within ESXD or before ED. Above two heuristics are to narrow down the candidate scope for fault locations based on the fault propagation effect.

Figure 3 shows fault propagation call trees for two test executions on a faulty version of *tot_info* program subject used in the experiment described in Section 4. In the call trees, each node is associated with a function execution and parent node calls its children nodes. The execution order among function executions is from the top to the bottom, with the earliest one in the top. An "ED" or "ESXD" in front of a function name denotes FVH differences. The names of the faulty functions that are located using our heuristics successfully are showed in bold font. In the first call tree, the fault is not propagated to the exit point of the faulty function so neither traditional system nor unit testing can expose that fault. In the second fault propagation call tree, the fault is propagated through two function boundaries and reaches the output statement in *main* function body, being observed by checking program outputs.

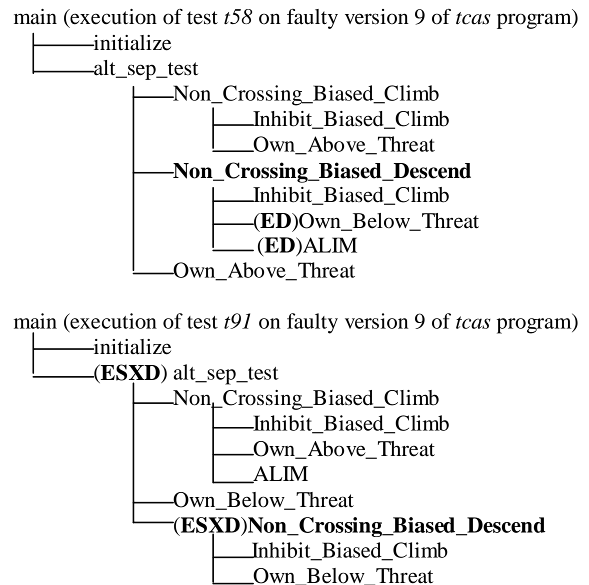

Figure 3. Fault propagation call trees

3.4. Spectra Evolution Accommodations

During program evolution, some program changes cause FVH differences without introducing regression

faults. Examples of this class of changes are summarized in Table 2 with respect to FVH differences they cause.

Table 2. Common Changes in Program Evolution

Category	Exemplary Changes
FVH-Insensitive Changes	Changes that do not cause FVH differences
Output-Insensitive Changes	Changes that make non-output variable values different but keep output variable values the same in function exit point.
Call Site Changes	Adding/removing function call sites
	Changing input values to function calls
Interface Syntax Changes	Adding/removing global variables
	Adding/removing function parameters
	Changing parameter/return types
	Changing function/parameter names
Incremental Interface Semantics Changes	Changes of a function to make function exit-point variable values different for subdomains of inputs and the subdomains can be formally specified
	Same as above except that the subdomains cannot be formally specified
Revolutionary Interface Semantics Changes	Changes of a function to make function exit-point variable values different for all inputs.

The first category comprises those changes that do not cause FVH differences, such as removing or inlining function calls, switching function call sites while preserving the meaning, adding new functions, etc. The second category comprises output-insensitive changes. These changes do not cause functional behavior differences as seen outside a function but cause differences of those non-output variable values at exit point, such as those values of argument variables. For this type of changes, those affected argument variables at exit point need to be filtered out manually before FVH is compared in our approach.

The third category comprises call site changes. These changes take place inside a function's body causing FVH differences for its callee invocations, but not causing the FVH differences in the variable value pair of that function. Examples of call site changes include when inputs to callee invocations are changed or when a function call site is added in a new version and that function already existed in the old version. Since these may well be symptoms of regression faults, this kind of change needs to be identified and filtered out by users.

The fourth category includes interface syntax changes. These changes affect the syntax of function interfaces, such as adding or removing function parameters or global variables, changing function or parameter names, changing parameter or return types, etc. However, these changes might not change the semantics of functions, which means the values of corresponding variables in FVH variable value pairs are still the same when some mappings between variable names in the old and new versions are specified.

The fifth category is incremental interface semantics changes. These changes cause semantic differences between a function in the old and new version for some subdomains of inputs. But for those inputs outside these subdomains, two versions of function executions still have the same variable value pair. Sometimes the subdomains can be formally specified using predicates, especially when those changes are specifically intended to enable the handling of an additional range of input values (such as better error checking). But sometimes the subdomains are difficult to formally specify, especially when the changes are for bug-fixing purposes and the exposure conditions of those bugs are not easy to capture (or are not perceived as worth specifying by the programmers). When the subdomains can be formalized by users, these predicates can be used to filter out those variable value pairs in the new version whose input variable values satisfy them in addition to those subsequently affected variable value pairs specified by users. When the subdomains are fault-inducing inputs and cannot be or are not formalized, they can be specified automatically as those fault-inducing FVH differences found by using our approach in previous regression testing cycle. The old version offers some reliable behaviors outside of the known fault-inducing input domain, which can be used to protect the program from being introduced new incidental errors during bug-fixing modifications.

The final category is revolutionary interface semantics changes. These changes make the behaviors of a function might be different between the old and new versions for all inputs. The affected function's variable value pairs and their subsequently affected variable value pairs need to be filtered out by users in the new version.

Mapping and filtering are two main mechanisms to handle those program changes that cause FVH differences but might not induce regression faults. Although this pre-processing before FVH comparison might require some manual efforts, several mechanisms can facilitate this process and reduce the cost. First, function declaration differences between the old and new versions attained by static analysis can guide the mapping and filtering that are related to interface syntax changes and some other changes. Second, the execution of a full-function-coverage or maximal-function-coverage test suite, instead of the whole test pool, can be used as trial-and-test pilot samples on the new version. Their FVH differences can guide the mapping and filtering interactively and iteratively, which can be generalized and applied to the rest of the test executions. Finally a visualizer (to be implemented in future work) could show FVH differences visually making user inspection and manipulation easier.

Finer-grained control of those change-affected variable value pairs is needed to specify the change impact boundaries— those function invocations that tightly enclose the FVH differences caused by fault-free program changes. We believe that while programmers or testers perform the mapping and filtering, they might have a deeper understanding of the changes made to the code and their impact from a dynamic and value-oriented perspective in addition to a static and structural perspective so that they might have more confidence in these code changes.

4. Experiment

4.1. Experiment Instrumentation

4.1.1. Tool prototype. We prototyped this approach to determine the practical utility. The basic idea is to take two versions of a program and to identify spectral differences between their executions on the same test. The Daikon [6] front end is used to instrument program code to collect data traces, which is the source from which we compute value spectra. We have only prototyped the key underlying mechanisms of our approach and users can directly edit a configuration file to manage filtering and mapping; this activity is supported by the declaration file that Daikon produces. Our tool can display value spectra differences in fault propagation call trees (as shown in Figure 3) and report potential fault locations. The scalability of data trace collection and value spectra comparison can be addressed by selective instrumentation. Only instrumenting the modified functions and those functions that are directly called by modified functions to collect data trace in the new version can approximate the results while reducing costs. But the same execution points in the old version still need to be instrumented and run to collect comparison data points a priori or posterior.

4.1.2. Subject programs. Seven C programs are used as subjects in the experiment. The researchers at Siemens Research created these seven programs with faulty versions and a pool of test cases [12]; these programs are popularly referred as the Siemens programs and are broadly used in regression testing empirical studies [10]. 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 only kept the faults that were detected by at least three and at most 350 test cases in the test pool. Table 3 shows basic information about these seven subject programs. The last column contains two numbers. The first number is the number of selected faulty versions in this experiment and the second number is the total number of faulty versions of each program.

Table 3. Subject programs in the experiment

Program	Function Number	LOC (Executable	Test pool size	Faulty versions
print_tokens	18	402	4130	7/7
print_tokens	19	483	4115	10/10
replace	21	516	5542	12/31
schedule	18	299	2650	9/9
schedule2	16	297	2710	9/9
tcas	9	138	1608	9/41
tot_info	7	346	1052	6/23

4.1.3. Test suites and versions. In the experiment, we use all the test cases in test pool for each program to reduce the potential threats to the validity introduced while selecting test cases for the experiment. For those programs with more than 10 faulty versions, we only pick those faulty versions in order from version 1 to make each selected version have at least one faulty function that has not yet occurred in previously selected versions. By doing so, the detailed results for each selected version can be presented in this paper given the limited space. Basically we conduct the experiment on base programs and each of their selected faulty versions.

4.2. Experiment Design and Results

4.2.1. Variables and Measures. The independent variables in the experiment are the subject programs and the selected faulty versions. We run all test cases in the test pool on the base version of each subject program to collect data trace information. And then we run all test cases on the selected faulty version to collect data trace information. For each combination of subject program and faulty version, the computed dependent variables measured in the experiment are as follows:

- *POV Diff*: The numbers of test case executions that exhibit POV differences.
- *FVH Diff*: The numbers of test case executions that exhibit FVH differences.
- *Cov*: The number of test case executions that cover the faulty statements in faulty version. The calculation results of *POV Diff*/*Cov* or *FVH Diff*/*Cov* imply the fault exposure probability of those test cases that cover the faulty statements by checking POV differences or FVH differences.
- H_1*_ok*, H_1*_all*, H_2*_ok*, and H_2*_all*: H_n*_all* is the number of test case executions that are eligible to use localization heuristic *n*. H_n*_ok* is the number of test case executions for which applying localization heuristic *n* successfully locates faults. The calculation

results of H_n_ok/H_n_all imply the success percentage of applying heuristic *n*.

For those faults that are in global data definition portion, we use the executable code that references the variables containing the faulty data to approximate the fault locations.

4.2.2. Threats to Validity. The threats to external validity primarily include the degree to which the subject programs, faults, and test cases are representative of true practice. This is an inherent issue with the Siemens programs, which are small and for which most of the faulty versions involve simple, one- or two-line manually seeded faults. Moreover, our experiment does 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.

The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype and Daikon front end might cause such effects. To reduce these threats, we manually inspected the FVH differences and FVH spectra on a dozen of traces for each program subject. The programs to be selected, the tests to be run, or the versions to be selected might cause the instrumentation effects. To control these effects, we apply our approach to each subject program and each test case. But for those programs with more than 10 faulty versions, we selected versions based on faulty function coverage simply to control the scale of the experiment.

One threat to construct validity is that our experiment makes use of the data trace collected during execution, assuming that these precisely capture the internal program states for each execution point. However, in practice the collected data traces leave out some information for engineering considerations. In next section, some related reasons that cause such imprecision are discussed.

4.2.3. Data and Analysis. Table 4-10 presents the results of the experiment for each subject. The first column shows the version number and the second one shows the abbreviations of faulty function names. When faults are in global data definitions (e.g. global variable value initializations in a header file), "*decl*" is showed instead. When faults are distributed among multiple functions, multiple functions are listed. The table cells of *POV Diff* and *FVH Diff* are shaded if *FVH Diff* is not greater than *POV Diff*. In the last two columns, H_n_ok and H_n_all is separated by a "/" mark. The table cells of H_n_ok/H_n_all are shaded if H_n_ok/H_n_all is less than one. Table 11 lists the statistical summary of the experiment results showing the overall success percentage of applying heuristics and fault exposure probability by checking POV differences or FVH differences.

Examining the results, it is observed that our approach increases the regression fault exposure probability about a factor of two compared to only checking program outputs. Both of our heuristics are effective in locating the faults and heuristic 1 is relatively more effective than heuristic 2. False positives are those reported faults that are indicated by spectra differences but are actually not faults. Although we do not quantitatively measure the percentage of false positives, while inspecting the reported locations of faults, we observed that when faulty functions are successfully reported for one test execution, usually there are few false positives reported at the same time.

For those versions whose faults are distributed in multiple functions, some test cases can expose all those faults in one test run, but some others can expose parts of them in one run. For those faults that lie in global data definitions, the reported fault locations are distributed in multiple places that reference those common faulty data. This might direct users' attention to the actual fault locations in the global data definitions.

For a few versions and tests, our fault localization heuristics could not precisely locate the faults. We inspected these traces carefully and identified some key reasons. For example, we faced some standard problems with floating point precision, unable to distinguish some minor floating point differences at the exit point of faulty functions (e.g. *tot_info* version 12). As another example, the traces capture the complete content for basic pointer-based data but only capture partial content for those complex pointer-based data structures such as linked list (e.g. some *schedule* and *schedule2* versions). In addition, the traces do not capture the external file stream pointed at by file descriptors (e.g. some *print_tokens* and *print_tokens2* versions). Since our value spectra comparison does not compare the structure among pointers, the pointer differences caused by faults are not detected when the contents pointed by the correct and wrong pointers are the same (e.g. some *replace* versions). Our approach identifies and utilizes FVH differences that do not consider the sequence order among variable value pairs. For some *print_tokens* versions, sometimes one infected variable value pair is not identified as ED because the variable value pair after infection is also present elsewhere in the old version when that corresponding function is executed multiple times in a test run. Although some locations other than the actual faulty functions are reported for these central reasons, we observe that most of the time they enclose the actual faulty functions quite tightly.

Table 4. print_tokens (test pool size: 4130)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	get_token	6	425	2648	425/425	
2	get_token	48	480	1594	480/480	
3	get_token	38	38	38	38/38	

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
4	decl	28	28	4070	25/26	3/3
5	get_token	150	1365	1400	1365/1365	10/10
6	decl	186	324	4070	138/172	198/215
7	numeric_cas	28	28	385	28/28	

Table 5. print_tokens2 (test pool size: 4115)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	get_token	240	249	3081		33/249
2	get_token	249	249	249	249/249	
3	get_token	33	758	759	758/758	
4	get_token	332	1431	3938	1431/1431	4/398
5	is_str_constant	173	173	173		173/173
6	is_num_constant	518	517	701		517/517
7	is_token_end	207	291	1429		291/291
8	is_token_end	256	256	3503		256/256
9	is_token_end	56	60	1429		60/60
10	is_str_constant	173	173	935		173/173

Table 6. replace (test pool size: 5542)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	dodash	68	295	2862	295/295	0//38
3	subline	130	142	4176	142/142	
6	locate	96	317	1745		317/317
7	in_set_2	83	88	1367		88/88
12	decl	309	5519	5519	5519/5519	0/88
14	omatch	137	303	1012	5/5	303/303
15	makepat	60	5009	5009		5009/5009
17	esc	24	52	52		52/52
19	getline	3	4204	4657	0/27	4177/4177
21	getline, etc.	3	5519	5519	5519/5519	33/38
22	getccl	19	1820	2890	1791/1820	0/18
27	in_pat_set	263	263	3913		263/263

Table 7. schedule (test pool size:2650)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	find_nth	4	7	1824		7/7
2	unblock_pro	210	1387	1592	386/1387	2/17
3	upgrade_pro	159	305	1358	305/305	1/15
4	upgrade_pro	294	622	1775	93/566	182/225
5	upgrade_pro	37	41	1317	41/41	
6	find_nth	4	7	1824		7/7
7	upgrade_pro unblock_pro	27	18	1831	18/18	
8	upgrade_pro	31	51	1342	13/51	3/6
9	main	23	23	2627	23/23	

Table 8. schedule2 (test pool size: 2710)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	upgrad_prio	65	311	1867	311/311	0/21
2	get_process	31	36	2667		36/36
3	get_process	34	133	2667		133/133
4	get_command	2	67	2580		67/67
5	new_job	32	64	2660		64/64
6	get_command	7	2575	2580		2575/2575
7	get_process	31	74	2667		74/74
8	put_end	60	97	2610		97/97
9	finish	0	0	2290		
10	enqueue	46	97	2660	97/97	0/58

Table 9. tcas (test pool size: 1608)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	Non_Crossing_BC	132	131	478		131/131
2	Inhibit_Biased_C	69	539	886		539/539
3	alt_sep_test	23	108	1578	108/108	
6	Own_Below_Th	12	14	601		14/14
7	initialize	36	1578	1578	8/8	1570/1570
9	Non_Crossing_BD	9	34	886	27/27	7/7
10	Own_Below_Th Own_Above_Th	14	16	886		16/16
13	decl	4	29	1578	29/29	
37	ALIM	97	373	564		373/373

Table 10. tot_info (test pool size: 1052)

V	Faulty Functions	POV Diff	FVH Diff	Cov	H1_ok/ H1_all	H2_ok/ H2_all
1	InfoTbl	158	158	158		158/158
2	main	10	10	963	10/10	
4	gcf	33	61	668		61/61
6	decl	46	46	1052	1/1	45/45
8	gser	199	199	253		199/199
12	LGamma	33	301	738		272/301

Table 11. Statistical Summary of Experiment Results

Items	Data	Items	Data
Mean(H1_ok/H1_all)	100%	Stdev (H1_ok/H1_all)	25.7%
Mean(H2_ok/H2_all)	100%	Stdev (H2_ok/H2_all)	37.6%
Mean(POV Diff/Cov)	3.5%	Stdev (POV Diff/Cov)	26.9%
Mean(FVH Diff/Cov)	10.3%	Stdev (FVH Diff/Cov)	39.1%

5. Related Work

Our approach is related to several different threads of research. Statistical summary of structural coverage information of a group of passing and failing tests on a faulty version is also used to guide fault localization [2][13]. These approaches can be ineffective when *Cov* is much greater than *POV Diff* in the faulty version or when execution information is provided for only a few tests. These approaches observe a set of passing and failing tests executions on a new version to locate faults in statement granularity; in contrast, our approach observes a single test execution on both the old and new versions to locate faults in function or code fragment granularity. Moreover, the infected variable values together with reported fault locations in our approach can better aid actual fault location determination and debugging process. Our

approach can complement existing structural approaches to improve fault localization effectiveness.

A dynamic slice 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 or tests [1]. It is used in locating the faulty statement given the infected variable values. Our approach complements this dynamic slicing approach since the results of our approach, the infected variable values, can provide inputs to the dynamic slicing approach.

Fault propagation has been broadly investigated in testing literature. The RELAY model is intended to understand how a fault may or may not cause a failure on execution of some test datum [20]. PIE (propagation, infection, and execution) analysis is used to assess the probability that under a given input distribution, if a fault exists in a code component, it will result in a failure [21]. This approach focuses on the estimation and analysis of fault exposure probability with the goal of generating or selecting test cases that propagate the faults to outputs. However, our approach focuses on regression testing fault exposure and localization by proactively exposing the faults even before they are (or even if they are not) propagated to outputs. Our approach can also offer an empirical way to study the fault propagation behaviors to complement existing analytic approaches.

Mutation testing is a specific form of fault injection that consists of creating different versions of a program by making small syntactic changes. Weak mutation testing requires that a test case causes a mutated version to compute a different value than original version does on at least one execution of mutated version in contrast to the strong mutation testing that requires mutated and original versions to produce different output [11]. The difference between our approach and traditional output-checking fault exposure approach is analogous to weak and strong mutation testing. Both weak mutation testing and our approach do not require the faults to be propagated to outputs. Weak mutation testing observes the exposure of hypothesized and seeded faults by a test for estimation purpose but our approach is observing the exposure of actual faults by a test for regression fault exposure and localization purpose. Moreover, weak mutation testing is primarily a unit testing technique, but our approach can be either system or unit testing technique by checking the internal behaviors inside a system or unit.

Runtime assertion checking [3][18] and dynamically inferred invariant detection [6][9] also check inside the black box. But these approaches do not focus on value infection caused by regression faults. Some regression faults might cause variable values to be different from the correct ones but still meet the assertions or invariants. Our approach can expose those faults that are exposed by these approaches when those violated assertions or invariants are defined in function entry or exit points. Again because of a different focus, assertion or invariant violations are less effective than our approach in locating faults since the infected variable values might be propagated across functions through certain number of assertions or invariants without violating them before they reach the points where an assertion or invariant is violated. The assertion or invariant violation locations might not be close to the fault locations.

The HERCULES deployment framework is proposed for upgrading components while keeping multiple versions of a component running [5]. The specific subdomain that a new version of a component correctly addresses is formally specified. For each invocation of the component, multiple versions of the component are run in parallel and the results from the version whose specified domain contains this invocation's parameters are selected. Handling of incremental interface semantics changes whose subdomain can be formally specified in our approach is similar to the HERCULES approach. However the HERCULES approach cannot handle the revolutionary interface semantics changes or the incremental interface semantics changes whose subdomain cannot be formally specified, which are common for bug-fixing and some other evolutions.

The relative debugging technique allows users to define a series of assertions between a reference program and a suspect program [19]. These assertions specify key data structures that must be equivalent at specific locations in two programs. According to these assertions, a relative debugger automatically compares the data structures and reports any differences while both versions are executed concurrently. It mainly aims at those data-centric scientific programs that are ported to, or rewritten for, another computer platform, e.g. a sequential language program being ported to a parallel language. Our approach can be applied in a broader scope of evolutions for more kinds of programs. Our approach is more effective in fault localization based on a fault propagation model and provides higher quality assurance and finer-grained controls over changes by checking value spectra instead of only checking user-specified data structures at user-specified locations.

Delta debugging algorithm isolates the relevant variables and values by systematically narrowing the internal program state differences between a passing and a failing test execution on a faulty version [22]. Our approach makes use of the internal program state differences between the test executions of the same test on an old version and a new version to expose regression faults and locate their locations. In their approach, the internal program states at only a few user-specified locations are captured by using memory graph [23], which are more accurate and complete than those captured by instrumenting source code with Daikon front-end in our approach. However, using memory graph in more than a

few locations is not scalable. Their approach can assist in debugging those faults besides regression ones, which are mainly targeted by our approach. However, their approach may require a large number of additional test runs and the symptoms for those test run failures be easily identified, e.g. program crash.

6. Concluding Remarks

We have developed a technique to augment and complement existing regression testing and to improve fault localization through comparing internal values across program versions. Checking inside the black box can improve testing effectiveness by increasing fault exposure probability in contrast to just checking the outputs of the black box. In regression system testing, our approach not only checks the input/outputs of the modified units inside the system black box, as traditional unit testing does, but also checks the internal variable values of the modified units in addition to the behaviors of their callees inside the unit black box. Our approach can expose faults without requiring them to be propagated outside the faulty function or across function boundaries to reach outputs. Our approach has other potential applications as well. For example, when the outputs of the system under regression testing are not convenient to check automatically, e.g. GUI outputs, our approach can be used to check internal program states other than the system outputs — these then act as proxies for the actual outputs. Moreover, the change impact boundaries in our approach can be used in dynamic impact analysis a posterior after changes are made complementing the existing dynamic impact analysis a priori before changes are made [14]. Finally our approach can be easily adapted to use in a component-based system or object-oriented system.

Programmers are reluctant to make changes to their code in the fear of breaking the existing code. Our approach can alleviate this fear by enabling them to understand and control the consequences of their changes in a fine-grained way. Therefore changes are exploited to improve the reliability and dependability during program evolution.

7. Acknowledgement

We thank Michael Ernst and the Daikon project members at MIT for their assistance in our installing and using Daikon tool. We are grateful for the assistance of Gregg Rothermel and Mary Jean Harrold in attaining the subject programs. This work was supported in part by the National Science Foundation under grant ITR 0086003. The authors wish to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

10. References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. and Exper.*, 23(6), pp. 589–616, June 1993.

[2] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and data flow tests. In *Proc. of IEEE Softw. Reli. Eng.*, pp. 143--151, 1995

[3] B. Andrews, Using Executable Assertions for Testing and Fault Tolerance. In *Proc. 9th IEEE Int'l Symp. Fault-Tole. Comp.*, pp. 102-105, 1978.

[4] T. Ball and J. R. Larus. Efficient path profiling. in *Proc. of Micro 96*, pages 46-57, Dec. 1996.

[5] J. E. Cook and J. A. Dage, Highly Reliable Upgrading of Components. In *Proc. Int'l Conf. Softw. Eng.*, pp. 203—212, 1999

[6] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proc. Int'l Conf. Softw. Eng.*, May 1999.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. on Softw. Eng. and Methodology*, 2(3):228–269, July 1993.

[9] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proc. Int'l Conf. Softw. Eng.*, May, 2002.

[10] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi. An Empirical Investigation of the Relationship Between Fault-Revealing Test Behavior and Differences in Program Spectra, *J. of Softw. Testing, Verifi., and Reli.* V. 10, no. 3, Sept., 2000.

[11] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. on Softw. Eng.,* 8(4):371--379, July 1982.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l Conf. Softw. Eng.*, pp. 191-200, May 1994

[13] J. A. Jones, M. J. Harrold, J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proc. Int'l Conf. Softw. Eng.*, 2002

[14] J. Law and G. Rothermel, Path Profile-Based Dynamic Impact Analysis, In *Proc. Int'l. Conf. Soft. Maint.*, Oct. 2002

[15] J. E.Payne, R. T. Alexander, and C. D.Hutchinson, Design-for-Testability For Object-Oriented Software, *Object Mag.*, pp. 35 – 43, July 1997

[16] S. P. Reiss and M. Renieris, Encoding program executions, In *Proc. Int'l Conf. Softw. Eng.*, pp. 221—230, 2001.

[17] 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 Softw. Eng. Notes*, 22(6):432-439, Nov. 1997.

[18] D. Rosenblum, Towards a Method of Programming with Assertions, In *Proc. Int'l Conf. on Softw. Eng.* pp. 92–104, 1992

[19] R. Sosic and D. Abramson. Guard: A Relative Debugger, *Softw. – Pract. and Expe.*, Vol. 27, No. 2, pp 185-206, Feb., 1997.

[20] M.C. Thompson, D.J. Richardson, and L.A. Clarke. An information flow model of fault detection. In *ACM Int'l Symp. Softw. Testing and Anal.*, pages 182--192, June 1993.
[21] J. Voas. PIE: A dynamic failure-based technique. *IEEE Tran. on Softw. Eng.*, pp. 717-727, Aug. 1992
[22] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *Proc. ACM Int'l Symp. Found. Softw. Eng.*, 2002.
[23] T. Zimmermann and A. Zeller. Visualizing memory graphs. In Proc. Int'l Dagstuhl Seminar on Soft Vis., pp. 191-204, 2002