An Empirical Evaluation of Test Case Filtering Techniques Based On Exercising Complex Information Flows

David Leon
Electrical Engineering & Computer Science Department
Case Western Reserve University
Cleveland, OH 44106
dzl@cwru.edu

Wes Masri
Computer Science Department
American University of Beirut
Beirut, Lebanon 1107 2020
wm13@aub.edu.lb

Andy Podgurski
Electrical Engineering & Computer Science Department
Case Western Reserve University
Cleveland, OH 44106
podgurski@case.edu

ABSTRACT
Some software defects trigger failures only when certain complex information flows occur within the software. Profiling and analyzing such flows therefore provides a potentially important basis for filtering test cases. We report the results of an empirical evaluation of several test case filtering techniques that are based on exercising complex information flows. Both coverage-based and profile-distribution-based filtering techniques are considered. They are compared to filtering techniques based on exercising basic blocks, branches, function calls, and def-use pairs, with respect to their effectiveness for revealing defects.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids, Monitors, Testing tools*; D.4.6 [**Operating Systems**]: Security and Protection – *Information flow controls.*

General Terms
Reliability, Experimentation.

Keywords
Test case filtering, dynamic information flow analysis, dynamic slicing, program dependences, software testing, observation-based testing.

1. INTRODUCTION
The idea of identifying and exercising *information flows* within a program is a long-standing theme in software testing research. Early work in this area focused on test data adequacy criteria that require exercising different kinds of *data flows* between program statements [30]; subsequent research addressed the more general concepts of *program dependence*s [20], which include both *data dependence*s and *control dependences*, and *program slices* [1], which are closely related. Podgurski and Clarke showed that for a program statement s_1 to affect the execution behavior of another statement s_2, there must be a (static) chain of data and/or control dependences connecting s_2 to s_1 [28]. Thus, if information flows from s_1 to s_2, then s_2 is directly or indirectly dependent on s_1. All of these concepts are important in testing because they reflect *interactions* between different program elements. Failures in deployed software are often associated with interactions that were not anticipated or tested by developers. However, basic software testing techniques such as *functional testing*, *statement coverage*, and *branch coverage* focus on exercising individual software features or program elements and may fail to exercise interactions that are critical to revealing certain program defects.

In principle, the conditions that cause a particular defect to trigger a failure may involve arbitrarily complex interactions between program elements. However, it is feasible to exercise only a limited number of different interactions during testing. The number of possible *n*-way interactions grows rapidly with *n*. It is often very difficult to create test data manually to exercise all interactions of a given type – even simple ones – and it follows from basic computability results that no general algorithmic solution to this problem exists. It is more tractable to instrument a program to *profile* interactions (record when they occur), to run the instrumented program on a test suite, and to *filter* the test cases based on the resulting profiles.

Test case filtering involves selecting a manageable number of tests to use from a large, existing test suite that contains redundant tests or is too large to use in its entirety [5][24].[1] Examples of such test suites include "legacy" test suites employed in regression testing and test suites obtained using inexpensive but possibly imprecise methods such as automatic test generation, capture of inputs in the field, or simulation. A subset of the test suite is selected based on an assessment of how likely it is to reveal any latent defects in the program under test. This assessment involves analysis of *profiles* of test executions. Various kinds of profiles can be used for this purpose, such as ones reflecting control flow, data flow, input or variable values, object states, event sequences, and timing.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

[1] We use the term "test case filtering" in preference to "test case selection" because the later is often used in the literature to refer to techniques for *creating* test cases.

There are two main reasons for filtering test cases: (1) to reduce the number of test cases that must be executed and (2) to reduce the number of test executions for which it is necessary to manually determine correct output or to audit (check) actual output. When the test cases to be filtered are automated and self validating, only reason (1) is relevant, and the filtering process does not involve executing test cases with the current version of the software under test. Reason (2) applies with respect to captured or simulated operational inputs and other test cases for which checking results entails manual effort. It is reasonable to execute such test cases with the software version to be tested, in order to collect execution profiles for use in filtering, since the cost of manually determining or auditing output typically dominates the cost of executing tests. Ideally, the filtering process is largely automated. An important special case of test case filtering is *regression test filtering* (often called *regression testing selection* in the literature) in which a regression test suite is filtered [31]. In regression test filtering, coverage profiles obtained when testing previous versions of the software may be used in filtering tests for use with the current version.

Some techniques for test case filtering are based on exercising data flows or program dependences [28][36], and test data adequacy criteria based on these notions can be recast as filtering criteria. The potential value of such techniques depends on the prevalence of software defects that are triggered only by certain kinds of information flows. This value must be balanced against the cost of collecting and analyzing profiles of such flows, which increases with the complexity of the flows. Filtering techniques based on exercising simple data flows, such as definition-use (def-use) pairs have been subjected to empirical evaluation (see Section 2). To our knowledge, the effectiveness of test case filtering techniques based exercising more complex information flows, such as ones involving a *combination* or *sequence* of dynamic data and/or control dependences, has not been evaluated empirically.

This paper reports the results of an empirical evaluation of several test case filtering techniques that are based on exercising complex information flows. Information flow profiles were obtained using a tool for dynamic information flow analysis and dynamic slicing that we developed [26]. Both coverage-based and profile-distribution-based filtering techniques are considered (see Section 3). The techniques are general-purpose; in particular, they are not specific to regression test filtering. They are compared to random sampling and to filtering techniques based on exercising basic blocks, branches, def-use pairs, and function calls.

Section 2 surveys graph-theoretic models of information flows proposed for use in software testing, and it describes the models employed in our empirical evaluation. Section 3 describes coverage-based and distribution-based test-case filtering techniques. Section 4 reports on our empirical study and its results. Section 5 surveys some additional related work. Section 6 presents our conclusions.

2. MODELING INFORMATION FLOWS

Software testing researchers have proposed graph theoretic models of several types of information flows and have used them to define testing techniques. In this section we present some illustrative examples of such models and indicate the models employed in our empirical evaluation. We assume the reader is familiar with control flow graphs and program dependences (See [26] for formal definitions of the terminology used here.)

Rapps and Weyuker define a family of test data adequacy criteria based on exercising "*du* (definition-use) paths" in a program's control flow graph. A *du-path* for a variable x is a path of the form uPv such that x is defined at u, x is used at v, and x is not defined along path P. Such a path demonstrates that v is *directly data dependent* on u with respect to x. Thus, Rapps and Weyuker's adequacy criteria exercise direct data dependences, which, together with direct control dependences, are the simplest form of information flow in programs. A number of papers describe empirical evaluations of one or more of Rapps and Weyuker's criteria [9][10][11][13][36].

Some software failures are associated with more complex interactions between program elements than those represented by direct data or control dependences. These interactions may correspond to a *sequence* or *combination* of direct dependences. Some proposed testing techniques are intended to exercise such interactions. For example, Ntafos proposed a family of test data adequacy criteria based on exercising chains of direct data flows (data dependences) called "*k-dr* interactions" [27]. Formally, a *k-dr interaction* in a program's control flow graph is a sequence of $k \geq 2$ vertices $\langle v_1, v_2, \ldots, v_k \rangle$ and a sequence of $k - 1$ variables $\langle x_1, \ldots, x_{k-1} \rangle$ such that for $i = 2, ..., k$, vertex v_i is directly data dependent on vertex v_{i-1} with respect to x_{i-1}. Ntafos's *required k-tuples* test data adequacy criterion is satisfied by a set T of test data if, among other conditions, T exercises each feasible *l-dr* interaction in a program's control flow graph at least once for $2 \leq l \leq k$.[2] Ntafos empirically evaluated only the required 2-tuples technique.

Laski and Korel proposed two test data adequacy criteria called "context coverage" and "ordered context coverage", which are based on exercising combinations of direct data flows [23]. These combinations are of two types. An (unordered) *definition context* for vertex v in a control flow graph G is a set of vertices $\{u_1, u_2, \ldots, u_n\}$ and a corresponding set of variables $\{x_1, x_2, \ldots, x_n\}$ such that there is a path Pv in G that can be decomposed for $i = 1, 2, ..., n$ into $X_i u_i Y_i v$, where the subpath $u_i Y_i v$ demonstrates that v is directly data dependent on u_i with respect to x_i. An *ordered definition context* for v is a sequence of vertices $\langle u_1, u_2, \ldots, u_n \rangle$ and a corresponding sequence of variables $\langle x_1, x_2, \ldots, x_n \rangle$ such that there is a path of the form $P_0 u_1 P_1 u_2 P_2 \cdots u_n P_n v$ in G, where for $i = 1, 2, ..., n$, the subpath $u_i P_i \cdots u_n P_n v$ demonstrates that v is data dependent on u_i with respect to x_i. Laski and Korel's *context coverage* (respectively *ordered context coverage*) criterion is satisfied by a set T of test cases for a program $\boldsymbol{P}$ if T exercises each feasible definition context (ordered definition context) in a $\boldsymbol{P}$'s control flow graph at least once. Laski and Korel did not evaluate their adequacy criteria empirically.

Clarke *et al* [3] showed that, with certain minor modifications, Ntafos's required *k*-tuples criteria and Laski and Korel's context coverage and ordered context coverage criteria *subsume* the family of data flow testing criteria defined by Rapps and

[2] We say that an information flow relationship defined in terms of a program's control flow graph is *feasible* if it is realized by a CFG walk whose corresponding sequence of program instructions is executed by some input(s).

Weyuker, in the sense that a test set satisfying the former criteria also satisfies the latter. This is true because the data flow relationships exercised by the former criteria are more general than those exercised by the latter criteria. Nevertheless, none of the aforementioned criteria models all of the kinds of information flows that can be associated with software failures. It is not difficult to prove that certain program failures can be triggered only by exercising arbitrarily complex information flows, e.g., arbitrarily long chains of data and/or control dependences.

These considerations lead naturally to the idea of defining test case filtering techniques in terms of even more general models of information flows in programs. Such models are employed in *program dependence analysis* [8], *information flow analysis* [4], and *program slicing* [34], which are closely related program analysis techniques that each support modeling of arbitrary combinations of *indirect* information flows between instructions or objects, where each indirect flow corresponds to a sequence of one or more direct data dependences and/or direct control dependences. Information flow analysis originated in the field of computer security [4]. It is used to determine if information stored in a sensitive variable or object can flow or actually has flowed to a variable or object that is accessible to an untrusted party. Program slicing is a debugging technique that seeks to identify the set of program statements – called a *slice* – that could be responsible for an erroneous program state that occurred at a particular location in a program. Information flow analysis and program slicing each have both static and dynamic variants. Both dynamic information flow analysis [26] and dynamic slicing [21], which involve analyzing runtime data and control dependences, are potentially much more precise than their static counterparts, because the outcomes of conditional branches become known at runtime.

Podgurski and Clarke described the semantic basis for the use of program dependence analysis in software testing, debugging, and maintenance [28]. They showed that the presence of a *syntactic dependence* (a chain of data and/or control dependences) between two statements is a necessary but not sufficient condition for one statement to affect the execution behavior of the other. They also argued that the number of tests required to adequately exercise all syntactic dependences can be impractically large, and they suggested that information about syntactic dependences might be useful for filtering test cases.

Thompson *et al* present an information flow model of fault detection, focusing on transfer of an incorrect intermediate state from a faulty statement to output [34]. Transfer occurs along information flow chains, where each link in the chain involves data dependence transfer or control dependence transfer.

Agrawal *et al* define a regression test filtering technique based on dynamic slicing [1]. The dynamic slices with respect to a program's output are computed for all test cases in its regression test suite. After the program is modified, the new program is run on only those test cases whose dynamic slices contain a modified statement. Agrawal *et al* also define a variant of this technique in which "relevant" program slices are computed. A *relevant program slice* is a dynamic program slice augmented to include certain predicates on which statements in the dynamic slice are *potentially dependent*. Agrawal *et al* did not empirically evaluate the effectiveness of either version of their technique.

In this paper, we empirically evaluate approaches to filtering test cases based on two closely related ways of characterizing complex information flows, namely: (1) tracing information flows between objects and (2) computing dynamic program slices. Since both forms of analysis produce large amounts of raw output, it is necessary to summarize information flows and slices using profiles that are more compact. The form of these profiles is described in Section 4.2.

3. FILTERING TECHNIQUES

In this section we describe two basic approaches to filtering test cases, which were compared in our empirical study. One approach calls for greedily selecting test cases to maximize coverage of program elements. The other approach calls for selecting test cases that span the profile-distribution of the original test suite.

3.1 Coverage-based Techniques

Coverage-based filtering techniques select test cases to maximize the proportion of program elements of a given type that are covered (executed). These techniques are based on the assumption that many software defects can be revealed simply by exercising such elements, regardless of other factors. To reduce testing costs, coverage-based filtering techniques attempt to cover as many elements as the original test suite with as few test cases as possible. This type of filtering is called *test suite minimization* in the regression testing literature [36][37]. Selecting a minimal-size, coverage-maximizing subset of a test suite is an instance of the *set-cover problem*, which is NP-complete but which admits a greedy approximation algorithm [14]. On each of its iterations, the greedy algorithm selects the test that covers the largest number of elements not covered by the previously selected tests. In the sequel, we will refer to this technique as *basic coverage maximization* to emphasize that code coverage is the basis for selecting test cases.

3.2 Distribution-based Techniques

Distribution-based filtering techniques select test cases based on how their execution profiles are distributed in the multidimensional profile space [5][6][24][25]. They identify features of the profile space, such as clusters, and use these to guide test selection. The profile space is defined by a *dissimilarity metric*, which is a function that for each pair of profiles outputs a real number representing their degree of dissimilarity. An example of a profile space is the *n*-dimensional space defined by applying the Euclidean distance metric to profiles that record basic-block execution counts for a program with *n* basic blocks. The tester may choose a dissimilarity metric emphasizing whatever aspects of the available profiles that he or she believes are most relevant to revealing defects. Typical dissimilarity metrics take the form of a (possibly weighted) sum of difference terms, in which there is a difference term for each profile feature (e.g., each execution count).

We consider two types of distance-based filtering techniques: cluster filtering and failure pursuit. *Cluster filtering* [5] is based on *automatic cluster analysis*. Cluster analysis is a multivariate analysis method for finding groups or clusters in a population of objects. Cluster analysis algorithms use a dissimilarity metric such as Euclidean distance or Manhattan distance to partition the population into clusters. Objects placed together in a cluster are

Table 1 – Number of unique profile features encountered during execution (unique execution counts) for the various types of profiles. *Combines MC, MCP, BB, BBE *and* DUP

	MC	*MCP*	*BB*	*BBE*	*DUP*	**ALL*	*IFP*	*SliceP*
javac	1,022	2,123	3,655	4,307	9,620	11,315	66,829	-
Javac$_{700}$	818	1,333	2,164	2,413	4,793	5,681	25,247	194,840
Xerces	361	690	1,725	1,982	3,812	4,519	6,547	84,565
JTidy	195	243	1,355	1,645	3,680	4,362	11,061	235,925

more similar to one another than to objects in other clusters. Cluster filtering uses cluster analysis to partition a set of tests into clusters based on the dissimilarity of their profiles. One or more test are selected for audit from each cluster or from particular clusters. A cluster filtering procedure is defined by a choice of clustering algorithm, dissimilarity metric, cluster count, and sampling method. An example of a sampling method is *one-per-cluster* (*OPC*) *sampling*, which calls for selecting exactly one test from each cluster. One-per-cluster sampling economically exercises each program behavior represented by a cluster, and it also favors the selection of *unusual* executions, which tend to be placed in isolated clusters. *Failure-pursuit sampling* is an adaptive extension of cluster filtering that is based on the observation that failed tests are often clustered together in small clusters [6]. Failure pursuit sampling calls for selecting the k nearest neighbors of any failures found by auditing the initial subset of tests. If any additional failures are found, each of their k nearest neighbors is selected, and so on, until no additional failures are found. (In the experiments reported in this paper, k = 5 is used.)

4. EMPIRICAL RESULTS

In this section we describe the subject programs and test suites used in our experiments. Then we describe the profile types that were used. Finally, we present and discuss the experimental results.

4.1 Subject Programs and Test Suites

In our experiments we applied test-case filtering techniques with different profile types on test suites for three Java programs: the *javac* Java compiler, version 1.3.1_02-b02 [17]; the *Xerces* XML parser, version 2.1 [38]; and the *JTidy* HTML syntax checker and pretty printer, version 3 [18].

javac was tested with the *Jacks* test suite [15], which tests compliance with the Java Language Specification [16]. The *Jacks* test suite comprises 3,140 tests among which 223 caused *javac* to fail.

Xerces was tested by using part of the XML Conformance Test Suite [39], which provides a set of metrics for determining conformance to the W3C XML Recommendation. There are 2000 tests in the XML TS contributed by several organizations such as Sun and IBM. We used 1663 tests in our experiments resulting in 10 failures. Note that we chose to exclude 337 tests because it was difficult to determine with certainty whether those tests were expected to pass or fail. *Xerces* was configured to check only the syntax and not the semantics of the input XML files, i.e., to simply check whether the files were *well-formed*.

JTidy was tested using 500 files downloaded from the Google Groups (groups.google.com) using a web crawler. Out of the 500, 5 were XML files and the rest were HTML files. *JTidy* failed on 24 of these test cases.

The defects causing the failures were investigated manually and the failures were classified into groups believed to have been caused by the same defect. For *javac*, 67 distinct defects were believed to have caused the 223 failures. (The failure classification for *javac* was done as part of previous work in order to validate an automated technique for classifying failures; see [29] for details). For *Xerces*, 5 distinct defects were believed to have caused the 10 failures. For *JTidy*, 5 distinct defects were identified, where 2 of them cause failures only in combination with another defect. Therefore, in our analysis of the *JTidy* profiles we treated each distinct combination as a defect on its own. This resulted in 6 defects for *JTidy*, 3 of which are combination defects.

4.2 Profiling

This section describes the profile types used in our experiments, then briefly describes the tools we built to generate them. Finally, for each combination of subject program and profile type, we show the number of unique profile features encountered during test suite execution (unique execution counts).

Program profiles identify the frequency of execution of certain program features that are thought to be relevant to whether executions succeed or fail. Such program features vary in complexity and the cost of profiling them varies accordingly. In our experiments, we profiled several program features of varying complexity. The profile types we used are listed and described below:

- Method calls (*MC*): Number of times each method was executed.
- Method call pairs (*MCP*): Number of times each method *M1* calls another method *M2*, for every combination of *M1* and *M2* for which this count is nonzero.

Table 2 – Results for conducting basic coverage maximization.

	Profile Type	% Tests Selected	% Defects	Selected/ Defects
javac	***MC***	1.63	13.7	5.59
	MCP	5.23	22.6	10.85
	BB	7.8	28.5	12.83
	BBE	10.05	32.8	14.40
	DUP	18.22	62.5	13.64
	ALL	19.3	63.2	14.30
	IFP	18.77	68.1	12.92
Javac$_{700}$	***MC***	5.21	15.05	10.10
	MCP	11.17	21.7	15.01
	BB	13.95	35.95	11.32
	BBE	17.17	46.12	10.85
	DUP	31.87	69.19	13.43
	ALL	32.66	72.85	13.07
	IFP	30.24	60.53	14.57
	SliceP	60.28	91.67	19.18
Xerces	***MC***	0.83	0	-
	MCP	3.57	24.22	49.160
	BB	9.78	40	81.51
	BBE	11.73	46.98	83.25
	DUP	15.94	60	88.58
	ALL	16.72	60	92.911
	IFP	10.43	70.28	49.51
	SliceP	20.63	100	68.8
JTidy	***MC***	2.71	33.33	6.79
	MCP	5.63	42.6	11.02
	BB	9.26	50	15.44
	BBE	11.45	83.33	11.45
	DUP	18.33	66.67	22.91
	ALL	19.04	83.33	19.04
	IFP	13.25	66.67	16.56
	SliceP	37.09	100.00	30.91

- Basic-blocks (*BB*): Number of times a basic block was executed.
- Basic-block edges (*BBE*): Number of times control flows from basic-block *B1* to basic-block *B2*, for every combination of *B1* and *B2* for which this count is nonzero.
- Def-use pairs (*DUP*): Number of times a statement *U* uses a variable defined by statement *D*, for each combination of *D* and *U* for which the count is nonzero.
- Information flow pairs (*IFP*): Number of times information from *x* flowed into *y* (as demonstrated by a sequence of dynamic data and/or control dependences leading from *y* to *x*), for each combination of *x* and *y* for which this count is nonzero. Here *x* and *y* are local variables, global (static) variables, or fields of a class instance. Note that when computing the *IFP* profiles the inter-procedural control dependences were not tracked.
- Slice pairs (*SliceP*): Number of times a statement s_1 appears in a slice computed for statement s_2, for each combination of s_1 and s_2 for which this count is nonzero.

Note that if the above descriptions are used directly, the resulting profiles contain a large amount of redundant information. For example, in the subject programs there are groups of basic blocks that were always executed together, and therefore their counts were the same in each execution. This redundant information was removed by replacing each group of profiles features that always had the same value (count) by a single feature. For example, when *Javac$_{700}$* was tested, close to 3.6 million distinct slice pairs were detected. These were replaced by 194,840 unique features. Table 1 shows for each program and profile type the number of unique profile features (unique counts) that were generated while running the test suites. For example, there were 84,565 different combinations of statements making up the *SliceP* profiles for *Xerces*, where each combination is made up of two statements s_1 and s_2, such that at least one slice computed at s_2 contained s_1. The column titled *ALL* shows the combined counts of *MC*, *MCP*, *BB*, *BBE* and *DUP*. This combined count is less than the sum of its components counts because of the removal of duplicates described above. As expected, Table 1 shows that profile types that characterize more complex program features have higher unique execution counts.

In order to generate the *IFP* and *SliceP* profiles we extended our existing tool for dynamic information flow analysis and dynamic slicing [26], which we call *DIFA*. The *DIFA* tool was originally designed to detect and debug insecure information flows in programs. It instruments the program's byte code classes and/or jar files in order to monitor program execution and computes slices and information flows as the program runs. For the purpose of our experiments, i.e., generating *IFP* and *SliceP* profiles, we added an optional capability that records the information flows and slices right after they get computed.

In order to generate the *MC*, *MCP*, *BB*, *BBE* and *DUP* profiles we built a specialized tool. Like the *DIFA* tool, this tool instruments the target byte code and then monitors program execution in order to profile the given program features.

Note that because of memory requirements, the 3,140 *SliceP* profiles generated for *javac* could not be analyzed to completion. Therefore, we present two sets of results for *javac*: one set based on all 3,140 *Jacks* tests that does not include *SliceP* and another set based on the first 700 *Jacks* tests that does include *SliceP*.

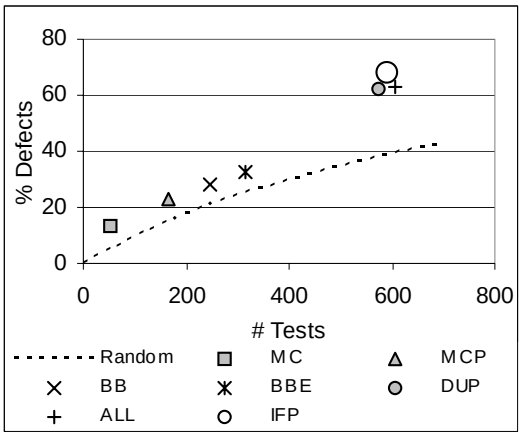

Figure 1 - Basic coverage maximization and random sampling results for *javac.*

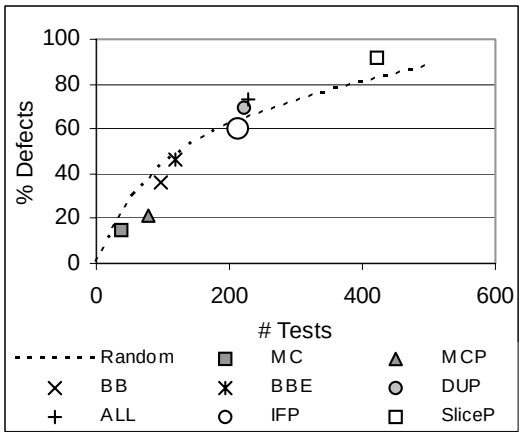

Figure 2 - Basic coverage maximization and random sampling results for *java*$_{700}$.

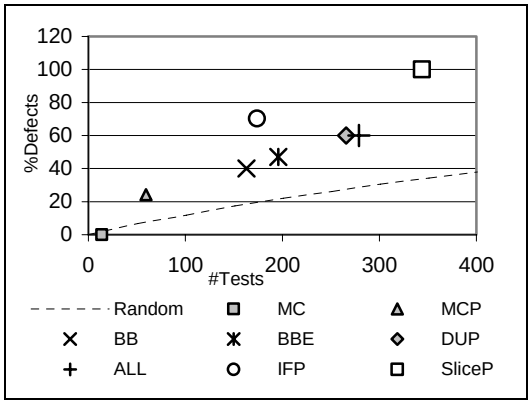

Figure 3 - Basic coverage maximization and random sampling results for *Xerces.*

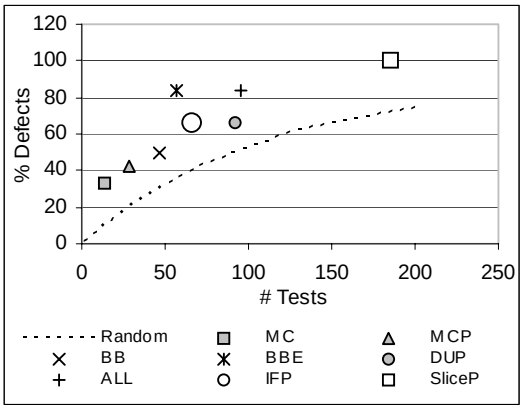

Figure 4 - Basic coverage maximization and random sampling results for *JTidy*.

The former will be referred to as the *javac* data set; the latter will be referred to as the *javac*$_{700}$ data set. For *javac*$_{700}$, 24 distinct defects were believed to have caused 128 failures. A matrix of the *SliceP* counts for this data set uses about 1GB of memory.

4.3 Basic Coverage Maximization Experiments

The results of conducting basic coverage maximization are shown Table 2. For example, in the case of *Xerces*/*SliceP*, the greedy selection algorithm demonstrated that no more than 20.63% of the original test suite was needed to exercise all of the dynamic slice pairs exercised by the original test suite, and these tests revealed all the defects revealed by the original test suite. Note that the greedy algorithm can sometimes encounter ties (multiple tests that each cover the maximal number of program elements not covered by previously selected tests). The way ties are broken affects the number of tests selected. To address this, we ran 1000 replications on each program/profile-type combination, first randomly shuffling the order of the tests. For each replication we recorded how many tests were selected and how many failures and defects were found. The data shown in Table 2 was obtained by averaging the results of 1000 different executions of the greedy coverage maximization algorithm, which explains why the columns showing the number of selected tests and the number of revealed defects contain fractions. Figures 1, 2, 3 and 4 compare variations of basic coverage maximization (based on different types of profiles) to random sampling with respect to each technique's average efficiency for revealing defects.

Figure 1 shows that with the *javac* data set, coverage maximization revealed defects more efficiently than random sampling did for all profile types. As expected, when the granularity of the elements being covered was finer, more tests were required and more defects were revealed. A significant jump in efficiency was observed when def-use pairs and

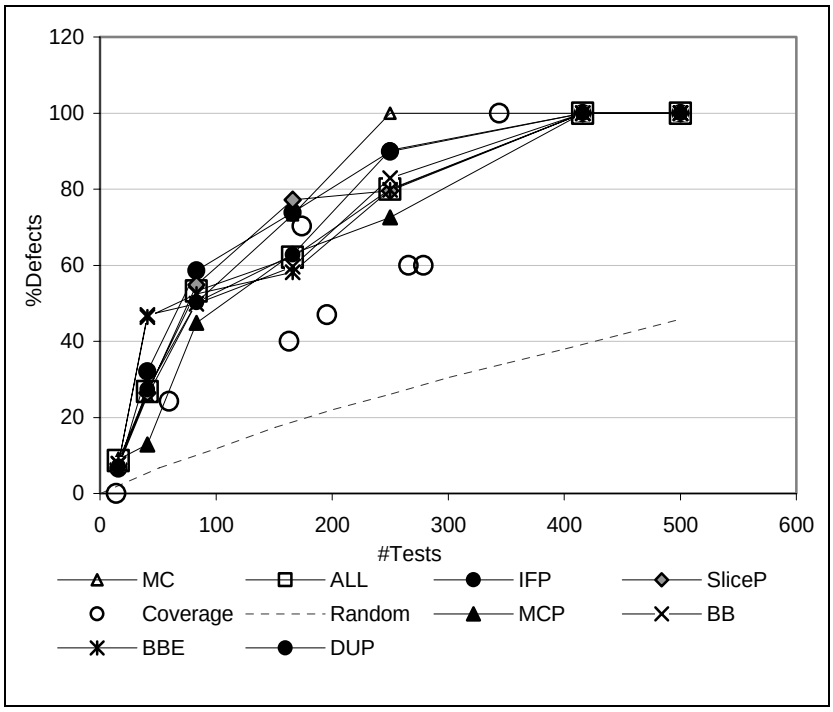

Figure 5 - OPC sampling, coverage maximization and random sampling results for *Xerces.*

information flow pairs were covered, although large numbers of tests were required to maximize coverage of these elements. Note that maximizing coverage of these elements was about as efficient as maximizing coverage of *ALL*, that is, of *MC*, *MCP*, *BB*, *BBE*, and *DUP* combined.

The results for $javac_{700}$ shown in Figure 2 differ considerably from the ones for *javac*. For example, basic coverage maximization was more efficient than random sampling only for *SliceP*, *DUP* and *ALL*. In addition, maximizing coverage of *DUP* and *ALL* revealed somewhat more defects than maximizing coverage of *IFP*. The good performance of random sampling in this case may be attributable to the large proportion of failures in this data set (0.18).

Figure 3 shows that with *Xerces*, basic coverage maximization performed better than random sampling except for with *MC* profiling. It also shows that maximizing *SliceP* coverage revealed all defects. Note however that maximizing *IFP* coverage revealed about 70% of the defects with only half as many tests.

Finally, Figure 4 shows that with *JTidy*, basic coverage maximization performed better than random sampling for all profile types. Maximizing *BBE* coverage revealed more defects than maximizing *IFP* coverage. This is possible because *IFP* profiles record only information flows between variables, while *BBE* profiles record branches involving basic blocks that do not include definitions of variables. Maximizing *IFP* coverage revealed about as many defects as maximizing *DUP* coverage though the former required fewer tests. Maximizing *SliceP* coverage revealed all defects but required 37% of all tests to be selected.

It should be noted that both of the two rightmost columns of Table 2 need to be considered when comparing the performance of one profile type to that of another. For example, for *javac* the ratio of number of tests selected to number of defects revealed is smallest for maximization of *MC* coverage (a favorable finding), but the percentage of revealed defects is unacceptably small. For *JTidy* on the other hand, maximizing *SliceP* coverage revealed all the defects but caused the aforementioned ratio to be considerably higher than for the other profile types.

4.4 Distribution-based Filtering Experiments

Previous experiments with cluster filtering and failure-pursuit sampling [5][6][24], which involved using only basic profiling techniques, suggested that it was most effective to use the *proportional dissimilarity metric* with *javac* and to use the *proportional-binary dissimilarity metric* with *Xerces* and *JTidy*. Hence, we did so in evaluating the usefulness of cluster filtering and failure-pursuit sampling with profiles reflecting complex information flows. The proportional metric compares two profiles based on the number of times profile features were exercised. It applies the *n*-dimensional Euclidean distance metric to profiles in which each feature value has been normalized to account for the range of values that the corresponding feature of the original profile took on. The proportional-binary metric is

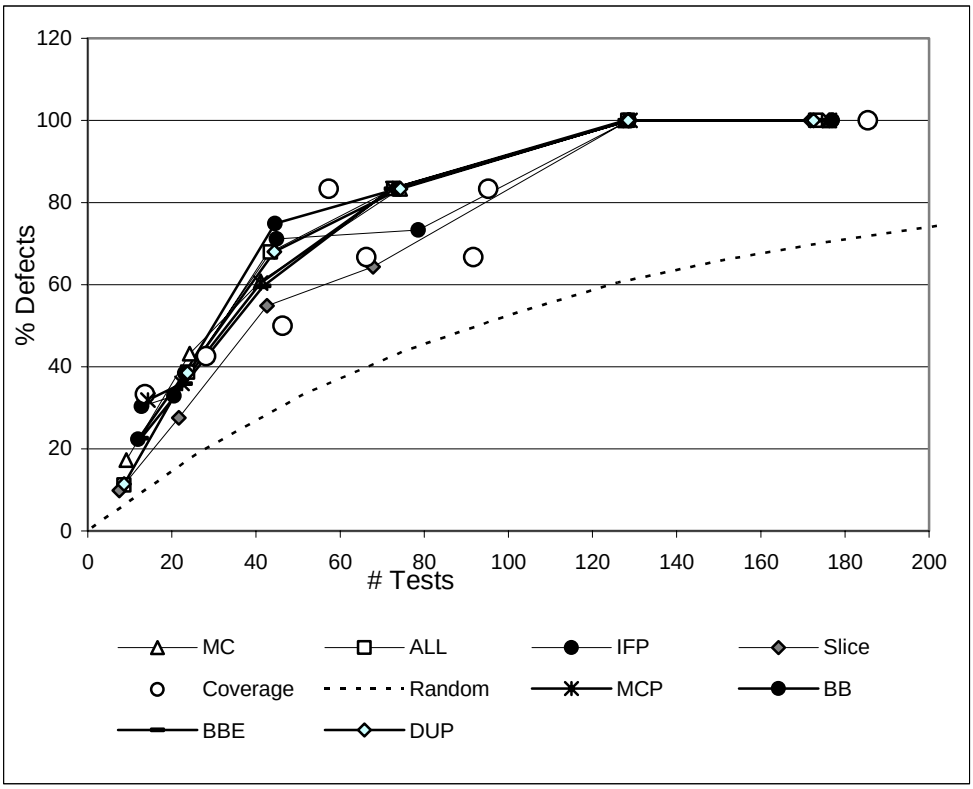

Figure 6 – Failure-pursuit sampling, coverage maximization and random sampling results for *JTidy*.

similar, but it replaces each feature of a profile with two features: one normalized as above and another that is binary and takes on the value 1 if the original value was positive. The second feature is included to increase the dissimilarity between executions that exercise particular program elements and those that do not.

In our experiments with cluster filtering and failure pursuit the *agglomerative hierarchical clustering* method [5] was used to cluster executions. The number of clusters was varied to correspond to the following percentages of the size of the test suite: 1%, 2.5%, 5%, 10%, 25%, and 30%. In the case of one-per-cluster sampling, for every program and profile type combination the experiments involved the following steps: (1) choosing *k*, (2) clustering executions, (3) randomly selecting a single test from each of the *k* clusters, and (4) recording the number of failures and defects revealed. The process of selecting the tests was replicated 1000 times and the results were averaged, which explains why the results presented include fractions. In the case of failure-pursuit sampling, steps (1)-(4) above were followed by the selection of additional tests as described in Section 3.2.

In our experiments, distribution-based filtering produced better results than random sampling did, but its performance did not depend substantially on the type of profile used. This is illustrated by Figure 5, which shows the one-per-cluster sampling results for *Xerces*, and by Figure 6, which shows the failure-pursuit results for *JTidy*. (For clarity, coverage maximization results obtained with different profile types are depicted with the same symbol.) The results suggest that with distribution-based filtering, using coarser profiles such as *MC* profiles may be as effective as using more complex profiles and more economical. Finally, it is not clear whether distribution-based filtering techniques generally perform better than coverage maximization when the number of tests selected with the latter is increased by using profiles of finer granularity.

5. ADDITIONAL RELATED WORK

Bates and Horwitz define test data adequacy criteria based on the program dependence graph, and they propose techniques based on program slicing to identify components of the modified program that can be tested with existing test cases and to identify components that may have been affected by the modification [2]. The adequacy criteria they define do not address indirect flows. Rothermel and Harrold present an approach to regression testing based on slicing, which uses a program dependence graph to identify changed def-use pairs [32]. Gupta *et al* present a similar approach that requires only partial data flow analysis following program changes and does not depend on a def-use history [12].

A number of empirical studies comparing different regression test selection or prioritization techniques in terms of their defect-detection effectiveness have been reported recently. (Note that the study reported in this paper does not address regression testing techniques in particular and does not make use of information about program changes.) Wong *et al* [36][37] studied the effectiveness of several test-suite reduction techniques. Graves *et al* examined the costs and benefits of several regression test selection techniques, including test suite minimization (greedy

coverage maximization), a dataflow technique, a safe technique, and random selection [11]. In separate studies, Elbaum, *et al* [7] and Rothermel *et al* [33] compared several test case prioritization techniques, including ones based on code coverage, estimated fault proneness, and other factors. Kim and Porter evaluated several regression test selection techniques and a technique and a prioritization technique of their own invention that exploits historical execution data [19]. None of the selection or prioritization techniques considered in the aforementioned studies address complex information flows.

Distribution-based filtering and prioritization techniques are examples of *observation-based testing*, which is described by Leon *et al* in [25] . Cluster filtering and several variants of it are presented and evaluated empirically by Dickinson *et al* [5]. Failure pursuit sampling is presented and compared empirically to cluster filtering by Dickinson *et al* [6]. Leon and Podgurski present an empirical comparison of four different techniques for filtering or prioritizing large test suites: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling [24]. None of this work addresses complex information flows.

6. CONCLUSIONS

We have empirically evaluated several test case filtering techniques that are based on exercising complex information flows, including both coverage-based and profile-distribution-based filtering techniques. They were compared empirically to filtering techniques based on exercising basic blocks, branches, function calls, and def-use pairs, with respect to their effectiveness for revealing defects. For three of the four data sets (*javac*, $javac_{700}$, and *JTidy*), maximizing coverage of information flows between objects required about as many tests and revealed about as many defects as maximizing coverage of definition-use pairs. On the remaining data set, the former technique required fewer tests and found more defects than the latter. Maximizing coverage of slice pairs revealed more defects than other coverage-based filtering techniques, at substantial additional cost in terms of test set and profile size. No clear difference in effectiveness was found between distribution-based filtering techniques (one-per-cluster and failure-pursuit sampling) and coverage maximization. Moreover, the effectiveness of the distribution-based techniques did not depend strongly on the type of profiling used. Thus, we found little evidence to justify the use of distribution-based filtering techniques based on exercising complex information flows. To confirm or refine these results, it will be necessary to conduct similar empirical studies with a variety of other subject programs and test sets.

7. REFERENCES

[1] Agrawal H., Horgan J., Krauser E., London S. Incremental Regression Testing. Proceedings of the IEEE Conference on Software Maintenance (Montreal, Canada, 1993).

[2] Bates, S. and Horwitz, S. Incremental Program Testing Using Program Dependence Graphs. 20th ACM Symposium on Principles of Programming Languages (January 1993), 384-396.

[3] Clarke, L. A., Podgurski, A., Richardson, D. J., and Zeil, S. J. A formal evaluation of data flow path selection criteria. IEEE Transactions on Software Engineering, Vol. 15, No. 11 (November 1989), 1381-1332.

[4] Denning D.E. and Denning P.J. Certification of programs for secure information flow. Communication of the ACM 20, 7 (1977), 504-513.

[5] Dickinson, W., Leon, D., and Podgurski, A. Finding failures by cluster analysis of execution profiles. 23rd Intl. Conf. on Software Engineering (Toronto, May 2001), 339-348.

[6] Dickinson, W., Leon, D., and Podgurski, A. Pursuing failure: the distribution of program failures in a profile space. 10th European Software Engineering Conf. and 9th ACM SIGSOFT Symp. on the Foundations of Software Engineering (Vienna, September 2001), ACM Press, 246-255.

[7] Elbaum, S., Malishevsky, A.G., and Rothermel, G. Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering 28, 2 (February 2002), 159-182.

[8] Ferrante J., Ottenstein K.J., and Warren J.D.. The Program Dependence Graph and its Use in Optimization. ACM Transactions on Programming Languages and Systems 9, 3 (October 1987), 319-349.

[9] Frankl, P. and Iakounenko, O. Further Empirical Studies of Test Effectiveness. 6th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, Nov. 1998), 153-162.

[10] Frankl, P. G. and Y. Deng. A Comparison of Delivered Reliability of Branch, Data Flow, and Operational Testing: A Case Study. 2000 International Symposium on Software Testing and Analysis (Portland, OR, August 2000), 124-134.

[11] Graves, T. L, Harrold, M. J., Kim, J. M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology 10, 2 (April 2001), 184-208.

[12] Gupta, R., Harrold, M. J., and Soffa, M. L. Program Slicing-Based Regression Testing Techniques. Journal of Software Testing, Verification, and Reliability 6, 2 (June 1996), 83-112.

[13] Harold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. Journal of Softare Testing, Verification, and Reliability, 10, 3 (September 2000).

[14] Hochbaum, D. S. (editor). Approximation algorithms for NP-hard problems. PWS Publishing, Boston, MA, 1997.

[15] Jacks, IBM, Jacks Project, www.ibm.com/developerworks/oss/cvs/jacks/, 2002.

[16] Java Language Specification, Sun Microsystems, java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, 2000.

[17] *javac*, Sun Microsystems Inc., Java™ 2 Platform, Standard Edition, java.sun.com/j2se/1.3/, 1995 – 2002.

[18] *JTidy*, jtidy.sourceforge.net, World Wide Web Consortium (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), 1998-2000.

[19] Kim, J. M. and Porter, A. A history-based test prioritization technique for regression testing in resource constrained environments. 2002 International Conference on Software Engineering (Orlando, FL, May 2002).

[20] Korel, B. The Program Dependence Graph in Static Program Testing. Information Processing Letters 24 (January 1987), 103-108.

[21] Korel B. and Laski J. Dynamic Program Slicing. Information Processing Letters 29 (October 1988), 155-163.

[22] Korel B. and Yalamanchili S. Forward Computation of Dynamic Program Slices. ISSTA (1994), 66-79.

[23] Laski, J. W. and Korel, B. A Data Flow Oriented Program Testing Strategy. IEEE Transactions on Software Engineering 9, 3 (May 1983), 347-354.

[24] Leon, D. and Podgurski, A. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. International Symposium on Software Reliability Engineering (Denver, CO, November 2003), 442-454.

[25] Leon, D., Podgurski, A., and White, L.J. Multivariate visualization in observation-based testing. Proceedings of the 22nd International Conference on Software Engineering (Limerick, Ireland, June 2000), ACM Press, 116-125.

[26] Masri, W., Podgurski, A., and Leon, D. Detecting and Debugging Insecure Information Flows. 15th. IEEE International Symposium on Software Reliability Engineering, ISSRE 2004. St. Malo, France Nov 2-5, 2004.

[27] Ntafos, S. C. On Required Element Testing. IEEE Transactions on Software Engineering 10, 6 (November 1984), 795-803.

[28] Podgurski A. and Clarke L. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. IEEE TSE, 16(9):965-979, September 1990.

[29] Podgurski. A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. and Wang, B. Automated support for classifying software failure reports. To appear in 2003 International Conference on Software Engineering (Portland, OR, May 2003).

[30] Rapps, S. and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering 11, 4 (April 1985), 367-375.

[31] Rothermel, G. and Harrold, M.J. A Safe, Efficient Algorithm for Regression Test Selection, 1993 Conference on Software Maintenance (September 1993), pages 358-367.

[32] Rothermel, G. and Harrold, M. J. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. 1994 International Symposium on Software Testing and Analysis (August 1994), 169-184.

[33] Rothermel, G., Untch, R., Chu, C., and Harrold, M.J. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27, 10 (October 2001), 929-948.

[34] Thompson, M. C., Richardson, D. J., and Clarke, L. A. An Information Flow Model of Fault Detection. International Symposium on Software Testing and Analysis (Cambridge, MA, June 1993), 182-192.

[35] Weiser M. Program Slicing. IEEE Transactions On Software Engineering 10, 4 (1984), 352-357.

[36] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P. Effect of test set size minimization and fault detection effectiveness. Software Practice and Experience 28, 4 (April 1998), 347-369.

[37] Wong, W. E., Horgan, J. R., Mathur, A. P., and Pasquini, A. Test set size minimization and fault detection effectiveness: a case study in a space application. 21st Annual International Computer Software and Applications Conference (Washington, D.C., August 1997), 522-528.

[38] *Xerces*. The Apache XML Project: xml.apache.org/xerces-j.

[39] XML Conformance Test Suite. www.w3.org/XML/Test