Towards the Prioritization of Regression Test Suites with Data Flow Information

Matthew J. Rummel
rummelm@allegheny.edu

Gregory M. Kapfhammer
gkapfham@allegheny.edu

Andrew Thall
athall@allegheny.edu

Department of Computer Science
Allegheny College

ABSTRACT

Regression test prioritization techniques re-order the execution of a test suite in an attempt to ensure that defects are revealed earlier in the test execution phase. In prior work, test suites were prioritized with respect to their ability to satisfy control flow-based and mutation-based test adequacy criteria. In this paper, we propose an approach to regression test prioritization that leverages the *all-DUs* test adequacy criterion that focuses on the definition and use of variables within the program under test. Our prioritization scheme is motivated by empirical studies that have shown that (i) tests fulfilling the *all-DUs* test adequacy criteria are more likely to reveal defects than those that meet the control flow-based criteria, (ii) there is an unclear relationship between *all-DUs* and mutation-based criteria, and (iii) mutation-based testing is significantly more expensive than testing that relies upon *all-DUs*.

In support of our prioritization technique, we provide a formal statement of the algorithms and equations that we use to instrument the program under test, perform test suite coverage monitoring, and calculate test adequacy. Furthermore, we examine the architecture of a tool that implements our novel prioritization scheme and facilitates experimentation. The use of this tool in a preliminary experimental evaluation indicates that, for three case study applications, our prioritization can be performed with acceptable time and space overheads. Finally, these experiments also demonstrate that the prioritized test suite can have an improved potential to identify defects earlier during the process of test execution.

1. INTRODUCTION

After a software system experiences changes in the form of bug fixes or additional functionality, a software maintenance activity known as regression testing can be used to determine if these changes introduced defects. The creation, maintenance, and execution of a regression test suite helps to ensure that the evolution of an application does not result in lower quality software. However, as noted by Beizer, many software development teams might choose to omit some or all of the regression testing tasks because they often account for as much as one-half the cost of software maintenance [1]. Moreover, the high costs of regression testing are often directly associated with the execution of the test suite [16]. Since some of the most well-studied software failures, such as the Ariane-5 rocket and the 1990 AT&T outage, can be blamed on the failure to test changes in a software system [9], many techniques have been developed to support efficient regression testing.

Regression test prioritization attempts to re-order a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with lower priority [4, 16]. By prioritizing the execution of a regression test suite, these methods hope to reveal important defects in a software system earlier in the regression testing process. Several current prioritization techniques have used either control flow-based (e.g., "branch coverage") or mutation-based test adequacy criteria [5, 17]. Yet, experiments have shown that (i) tests that fulfill the *all-DUs* test adequacy criteria are more likely to reveal defects than those that satisfy the control flow-based criteria [11], (ii) there is an unclear relationship between *all-DUs* and mutation-based criteria [7], and (iii) mutation-based testing is significantly more expensive than testing that relies upon *all-DUs* [4, 7].

In light of these observations, there is a clear need for a prioritization approach that has the potential to be more effective than those techniques that rely upon control flow and mutation-based adequacy and is less costly than the mutation-based schemes. In this paper, we focus on the definition and use of program variables by employing the *all-DUs* test adequacy criteria to prioritize a regression test suite. Our approach is implemented in a tool, called *Kanonizo*, that prioritizes JUnit test cases and subsequently evaluates the effectiveness of these prioritizations. In summary, the important contributions of this paper are as follows:

1. The description of a technique that uses data flow information to prioritize a regression test suite.

2. The formal statement of the algorithms and equations that are used to instrument the program under test, monitor coverage during test suite execution, and calculate test adequacy.

3. An empirical evaluation of (i) the time and space overheads of our approach and (ii) the effectiveness of the resulting prioritized test suites.

SAC'05, March 13–17, 2005, Santa Fe, New Mexico.

Test Case	Faults				
	f_1	f_2	f_3	f_4	f_5
T_1			×	×	
T_2	×	×			
T_3	×	×	×		
T_4			×	×	×
T_5		×	×		

Figure 1: The Faults Detected by a Test Suite.

2. MOTIVATION

Normally, a software tester is not aware of the defect locations within the program under test. However, to make our discussion more concrete, suppose that program P is tested with test suite T and a priori knowledge of the faults within P is available. Figure 1 shows an example of a simple test suite that was initially described in [12]. In this example, there are five faults that are revealed by certain test cases. In this test suite, some tests isolate more faults than other test cases. In particular, tests T_3 and T_4 are able to reveal all of the faults within the program under test. Intuitively, it would be better if these two test cases are executed before the other tests within the suite. Over the entire execution of the test suite, the test tuple $\sigma_1 = \langle T_1, T_2, T_3, T_4, T_5 \rangle$ yields a smaller weighted average percentage of isolated defects than the tuple $\sigma_2 = \langle T_3, T_4, T_1, T_2, T_5 \rangle$ (i.e., σ_2 detects faults faster than σ_1). Alternatively, if the goal is to identify all of the defects in the program while executing the smallest number of tests, the second tuple σ_2 fulfills this objective better than the first tuple σ_1.

Since the existence of a priori knowledge about the location of faults within the program under test is unlikely, regression test suite prioritization algorithms must use a proxy for this complete knowledge. Current regression test suite prioritization algorithms are motivated by the empirical investigations of the effectiveness of test adequacy criteria which indicate that tests that are not highly adequate are often less likely to reveal program defects [10, 11]. In light of the correlation between low adequacy test suites and the decreased potential to reveal a defect [11], a prioritization algorithm might chose to execute highly adequate tests before those with lower adequacy. Of course, since highly adequate tests are not guaranteed to always reveal the most defects, prioritization schemes can still fail to produce an optimal ordering of the tests [11, 12].

3. TERMINOLOGY

Problem 1 characterizes the regression testing problem in a fashion used in [15]. Furthermore, Definition 1 defines the test suite T that is subject to prioritization. A test suite contains a tuple of tests $\langle T_1, \ldots, T_r \rangle$ that execute in a specified order. We require that each test is *independent* so that we can guarantee that for all $\gamma \in [1, r]$, $\Delta_\gamma = \Delta_0$ and thus there are no test execution ordering dependencies [12, 13]. This requirement enables our prioritization algorithm to re-order the tests in any sequence that maximizes the suite's ability to isolate defects. The assumption of test independence is acceptable because the JUnit test execution framework provides `setUp` and `tearDown` methods that execute before and after a test case and can be used to clear application state.

Problem 1. (Rothermel and Harrold [15]) Given a program P, its modified version P', and a test suite T that was used to previously test P, find a way to utilize T to gain sufficient confidence in the correctness of P'. □

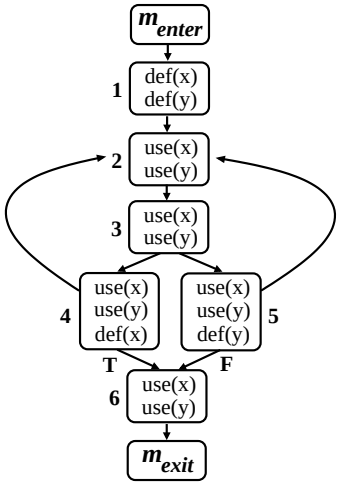


Figure 2: Example of a Control Flow Graph.

Definition 1. A test suite T is a triple $\langle \Delta_0, \langle T_1, \ldots, T_r \rangle, \langle \Delta_1, \ldots, \Delta_r \rangle \rangle$, consisting of an initial external test state, Δ_0, a test case sequence $\langle T_1, \ldots, T_r \rangle$ for state Δ_0, and expected external test states $\langle \Delta_1, \ldots, \Delta_r \rangle$ where $\Delta_f = T_f(\Delta_{f-1})$ for $f = 1, \ldots, r$. □

Data flow testing is intuitively motivated by the realization that a program will only be able to determine if the definition of a variable is correct if it subsequently uses the variable. The occurrence of a variable on the left hand side of an assignment statement is called a *definition* of this variable. The *use* of a variable occurs when it appears on the right hand side of an assignment statement or in the predicate of a conditional logic statement or an iteration construct [11]. Figure 2 provides an intuitive depiction of a control flow graph (CFG) for a method m that will be used during our discussion of data flow-based test adequacy.

This paper focuses on intra-procedural definitions and uses that occur in a single program method. For example, the statement `x = x + y` uses the variables `x` and `y` and then subsequently defines `x`. The node labeled with a "4" in Figure 2 would represent the definition and uses of program variables for the statement `x = x + y`. In order to formally define the *all-DUs* criterion, we view a method in the program under test as a control flow graph $G = (N, E)$ where N is the set of CFG nodes and E is the set of CFG edges.

We also define a *definition clear path* for variable *var* as a path in a CFG $\langle n_\rho, \ldots, n_\tau \rangle$, such that none of the nodes $n_\rho, \ldots, n_\tau$ contain a definition or undefinition of program variable *var* [8]. Next, we define the *def-use association* as a triple $\langle n_d, n_u, var \rangle$ where a definition of variable *var* occurs in node n_d and a use of *var* occurs in node n_u [11]. For example, $\langle n_1, n_4, x \rangle$ is one of the sixteen def-use associations within the CFG provided in Figure 2.[1] A *complete path* is a path in a method's control flow graph that starts at the CFG's entry node and ends at its exit node [8].

A complete path π_{var} *covers* a def-use association if it has a definition clear sub-path, with respect to *var* and the method's CFG, that begins with node n_d and ends with node n_u [8]. Therefore, the complete path

[1]Due to space constraints, we do not list all of the def-use associations. Both `x` and `y` are involved in eight associations.

```
Algorithm InstrumentAndEnumerate(P)
Input: Program Under Test P
Output: Instrumented Program Under Test P_TCM;
        Sets of Test Requirements R(P)
1.  P_TCM ← P, R(P) ← ∅
2.  for m ∈ methods(P_TCM)
3.      do R(m) ← ∅
4.          G_TCM ← GetControlFlowGraph(P_TCM, m)
5.          for var ∈ U_var(m)
6.              do for n_u ∈ UseLocations(var, m)
7.                      do for n_d ∈ ReachingDefinitionLocations(var, m)
8.                              do R(m) ← R(m) ∪ {⟨n_d, n_u, var⟩}
9.                                 n^d_track ← CreateDefineTrackingStatement(var)
10.                                N_TCM ← N_TCM ∪ {n^d_track}
11.                                E_TCM ← E_TCM ∪ {(n^d_track, n_d)}
12.                                for n_p ∈ pred(n_d)
13.                                    do E_TCM ← E_TCM ∪ {(n_p, n^d_track)}
14.                                n^u_track ← CreateUseTrackingStatement(var)
15.                                N ← N ∪ {n^u_track}
16.                                E_TCM ← E_TCM ∪ {(n^u_track, n_u)}
17.                                for n_p ∈ pred(n_u)
18.                                    do E ← E ∪ {(n_p, n^u_track)}
19.         R(P) ← R(P) ∪ {R(m)}
20.         SetControlFlowGraph(G_TCM, P_TCM, m)
21. return P_TCM, R(P)
```

Figure 3: The Instrumentation and Test Requirement Enumeration Algorithm.

$\langle entry_m, n_1, n_2, n_3, n_4, n_6, exit_m \rangle$ would cover the def-use association $\langle n_1, n_4, x \rangle$ for variable x. Finally, Definition 2 defines the *all-DUs* test adequacy criteria. In this definition, we use $U_{var}(m)$ to denote the universe of program variables within the method under test m. Since our CFGs include nodes to define temporary variables that represent the formal parameters of methods and global variables [3], $U_{var}(m)$ contains these temporary variables and all local method variables. Intuitively, a test suite T is more adequate, with respect to *all-DUs*, if it covers more of the def-use associations within the program under test.

Definition 2. A test suite T for method m's control flow graph $G = (N, E)$ satisfies the *all-DUs* test adequacy criterion if and only if for each def-use association $\langle n_d, n_u, var \rangle$, where $var \in U_{var}(m)$ and $n_d, n_u \in N$, there exists a test in T to create a complete path π_{var} in G that covers the association. □

4. EVALUATION METRICS

All of the evaluation metrics used in this paper assume an a priori knowledge of the faults that exist within the program under test. During the empirical evaluation of regression test suite prioritization, techniques that create a prioritized test suite can be evaluated based upon the weighted average of the percentage of faults detected over the life of the test suite, or the *APFD* [5]. Intuitively, if one approach to prioritization produces test suite orderings with higher *APFD* values than another technique, then the first prioritization scheme should be preferred. Since the *APFD* metric was used in early studies of regression test suite prioritization techniques and because it can still be used as a basis for more comprehensive prioritization approaches that use cost-benefit thresholds [5], this paper uses it as one metric to evaluate our prioritizations.

If we use a variant of the notation established in [6, 12] and we have a test suite T with r total tests and a total of g faults within program under test P, then Equation (1) defines the $APFD(T, P) \in [0.0, 1.0]$ [5]. We use $reveal(i, T)$ to denote the position within T of the first test that reveals fault i. If we examine the test orderings proposed in Section 2, then the $APFD$ for a test suite T_1 that contains σ_1 is $APFD(T_1, P) = 1 - .4 + .1 = .7$ and the $APFD$ for T_2 that uses test tuple σ_2 is $APFD(T_2, P) = 1 - .2 + .1 = .9$. Finally, the percentage of the test suite that must be executed in order to find all of the defects, or $PTR(T, P) \in [0.0, 1.0]$, is defined in Equation (2) [4]. In this equation, we use r_g to denote the number of test cases that must be executed to find all g program defects for a test suite with a total of r test cases. An examination of the same T_1 and T_2 shows that $PTR(T_1, P) = \frac{4}{5}$ and $PTR(T_2, P) = \frac{2}{5}$.

$$APFD(T, P) = 1 - \frac{\sum_{i=1}^{g} reveal(i, T)}{rg} + \frac{1}{2r} \qquad (1)$$

$$PTR(T, P) = \frac{r_g}{r} \qquad (2)$$

5. PRIORITIZATION APPROACH

In order to prioritize a regression test suite, the set of test requirements that are called for by the *all-DUs* test adequacy metric must be enumerated. Furthermore, instrumentation that records the test requirements that were covered during the test execution phase must be introduced into the program under test. The *InstrumentAndEnumerate* algorithm in Figure 3 performs two key functions: (i) calculates the set of test requirements for program P, denoted $R(P)$, that are necessitated by *all-DUs* and (ii) introduces test coverage monitoring instrumentation into P in order to produce P_{TCM}. The algorithm analyzes each method within P and

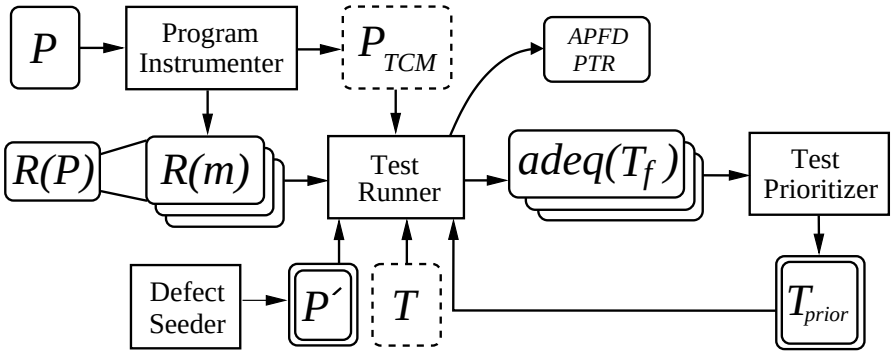


Figure 4: *Kanonizo*'s Approach to Prioritization.

performs intra-procedural data flow analysis to identify each def-use association within each method m. To this end, $R(P)$ is defined to be a set of the sets $R(m)$ that exist for each method m within the program under test.

In the *InstrumentAndEnumerate* algorithm, $pred(n_\tau)$ is used to denote the set of predecessor nodes for node n_τ. Using the node predecessor information, definition and use "tracking statements" are introduced before each define and use location within m's control flow graph G. After all coverage monitoring statements have been properly inserted, the initial control flow graph G for method m is replaced with the instrumented CFG denoted G_{TCM}. Upon completion of the algorithm in Figure 3, P_{TCM} can be used during the execution of T in order to produce $R_c(m)$, the set of covered test requirements, for each method m that was tested by a test case.

When a test case has caused the coverage of both n_d and n_u for variable var, the tracking statements introduced into P_{TCM} add the def-use association $\langle n_d, n_u, var\rangle$ into $R_c(m)$. Once the test suite has executed, it is possible to calculate the adequacy for each test case $T_f \in \langle T_1, \ldots, T_r\rangle$. To this end, we use $methodsTested(T_f) = \{m_1, \ldots, m_h\}$ to denote the set of methods that are tested by test T_f. For an arbitrary $m_k \in methodsTested(T_f)$, our test coverage monitoring instrumentation reports $R_c(m_k)$. Equation (3) defines the cumulative adequacy of a test case as the ratio between the number of covered test requirements and the total number of test requirements, for all of the methods under test. In this paper, we use $adeq(T_f)$ to prioritize a regression test suite according to its ability to satisfy the *all-DUs* criterion. For example, if T_f enters method m in Figure 2, executes the true branch of node n_3, and exits method m, then we have $adeq(T_f) = \frac{7}{16} = 43.75\%$.

$$adeq(T_f) = \frac{\sum_{k=1}^{h} |R_c(m_k)|}{\sum_{k=1}^{h} |R(m_k)|} \quad (3)$$

Figure 4 depicts our approach to the prioritization of a regression test suite. After the set of test requirements has been computed for each method within P and P_{TCM} has been produced, the test runner can execute T and produce $adeq(T_f)$ for each T_f. In Figure 4, the inputs with a dashed border are used during the first invocation of the test runner. Upon completion of the first execution of the test suite, the test adequacy values are input into the test prioritization module that re-orders the test suite according to the adequacies. *Kanonizo* also includes additional modules that are used during the experimental evaluation of the prioritized test suite T_{prior}. To this end, a defect seeder is responsible for placing faults into the methods within program P that are tested by T. For more details about the faults that were seeded during experimentation, please refer to Section 6. Finally, the inputs with the double border are used during the second execution of the test runner and the $APFD$ and PTR metrics are calculated in order to evaluate the effectiveness of the prioritization created by *Kanonizo*.

6. EXPERIMENT GOALS AND DESIGN

We used an implementation of our data flow-based prioritization scheme to conduct two types of experiments: (i) the determination of the time and space overheads incurred during the execution of *InstrumentAndEvaluate* and the use of the test runner and (ii) the evaluation of the metrics described in Section 4 in comparison to randomly ordered test suites. All experiments were conducted on a GNU/Linux workstation with kernel 2.4.18-14smp, dual 1 GHz Pentium III Xeon processors, 512 MB of main memory, and a SCSI disk subsystem. We used Soot 1.2.5 [18] to implement a tool that can calculate the intra-procedural def-use associations and introduce test coverage monitoring instrumentation into all of an application's methods. Our tool uses the Jimple intermediate representation [18] to analyze the methods in the candidate applications. *Kanonizo* also uses a modified version of the JUnit test automation framework in order to execute the test cases and report the metrics of test case adequacy, $APFD$, PTR, and the time and space overhead.

Since the calculation of the metrics described in Section 4 requires a priori knowledge of the faults within P, we manually seeded the program under test with defects. We examined the mutation operators that are currently available for the Java programming language (e.g., [2, 14]) and applied these operators to the selected case study applications. For example, we inserted defects associated with the values of programmer defined constants by changing `INVALID_DEPOSIT = 5.00` to `INVALID_DEPOSIT = 9.0`. We also mutated the relational operators used in conditional logic statements and the arithmetic operators used in assignment statements. In certain methods of the program under test we removed lines of code or changed the return value.[2] Since the $APFD$ metric requires the position of the

[2]Due to space constraints, we do not describe every error that was seeded in the program under test. We seeded the defects without regard to the likelihood of their being revealed by a test that was highly adequate according to the *all-DUs* metric. Seeded faults did not modify the previously calculated adequacies.

Program	Time (ms)	Space (bytes)
Bank	3,210	1,084,648
Identifier	3,351	2,170,801
Money	9,176	4,984,648

Figure 5: *InstrumentAndEnumerate* Overheads.

first test case in suite T to expose fault i in order to calculate $reveal(i, T)$, we determined this before experimentation. To this end, each fault was seeded in isolation from all other defects, each test suite was executed for its application, and the fault revealing test cases were noted.

Our experiments focused on the prioritization of test suites for three applications written in the Java programming language: `Bank`, `Identifier`, and `Money`. Since we did not modify the JUnit test suite that was created for each application, it was not possible to control the scope of a test case. Thus, a "test" for one case study application might perform considerably more or less testing operations than a test for another program. The `Bank` application is a modification to a class created by Cay Horstmann that was changed by an undergraduate computer science student. The application simulates the possible actions that can be performed on a typical bank account system such as checking the balance, making a deposit, and making a withdrawal. It contains 1 class, 53 def-use associations, 5 methods and 7 test cases. Since this application is the smallest of those used, four faults were seeded into the program and two randomized orderings were run and compared with the prioritized suite.

The `Identifier` application was also written by undergraduate computer science students. This case study program uses the Flyweight and Visitor design patterns to recognize reserved and non-reserved words which are passed to a text processing system. `Identifier` contains 3 classes, 81 def-use associations, 13 methods and 11 test cases. For this application, we also used two random test suite orderings in comparison to the prioritized suite in a two phase experiment. In the first phase, three faults were seeded into the program and the ordered suites were executed. This was followed by a second phase where three additional faults were added and the same three types of ordered test suites were executed again. This approach resulted in a total of six seeded defects in the second execution of the tests.

Finally, the `Money` application is a program that is provided with the JUnit framework written by Erich Gamma and Kent Beck. This programs simulates the actions of adding and removing specified types of currency into and out of bags of money. It contains 3 classes, 302 def-use associations, 33 methods and 21 test cases. In our experiments, `Money` is the largest application of those tested in terms of both the number of def-use associations and test suite size. As in the previous experiments, two random orderings were compared to the prioritized suite. However, in this experiment the suites were executed three times with three additional faults being added at each stage of test execution. This approach resulted in a total of nine seeded faults in the third execution of the test cases.

7. RESULTS ANALYSIS

Figure 5 shows the time and space overheads associated with running the *InstrumentAndEnumerate* algorithm on the three case study applications. These results clearly indicate that the time and space overheads increase as the size of the application increases. However, it is important to observe that all of these overheads are relatively minimal since the largest application only incurs a time overhead of $9,176$ ms and $4,984,648$ bytes. Thus, for the selected case study applications, the *InstrumentAndEnumerate* algorithm that is at the core of our prioritization approach can be executed in an efficient fashion.

Our experiments revealed no noticable difference in the time overheads associated with test execution. For example, the execution of the test suite against the instrumented and non-instrumented version of `Money` both took approximately 64 ms. Finally, the execution of the tests on the instrumented `Money` only required the consumption of $102,936$ bytes more than the non-instrumented ones. For our case study applications, we can conclude that the test coverage monitoring instrumentation does not introduce any significant overheads into test suite execution.

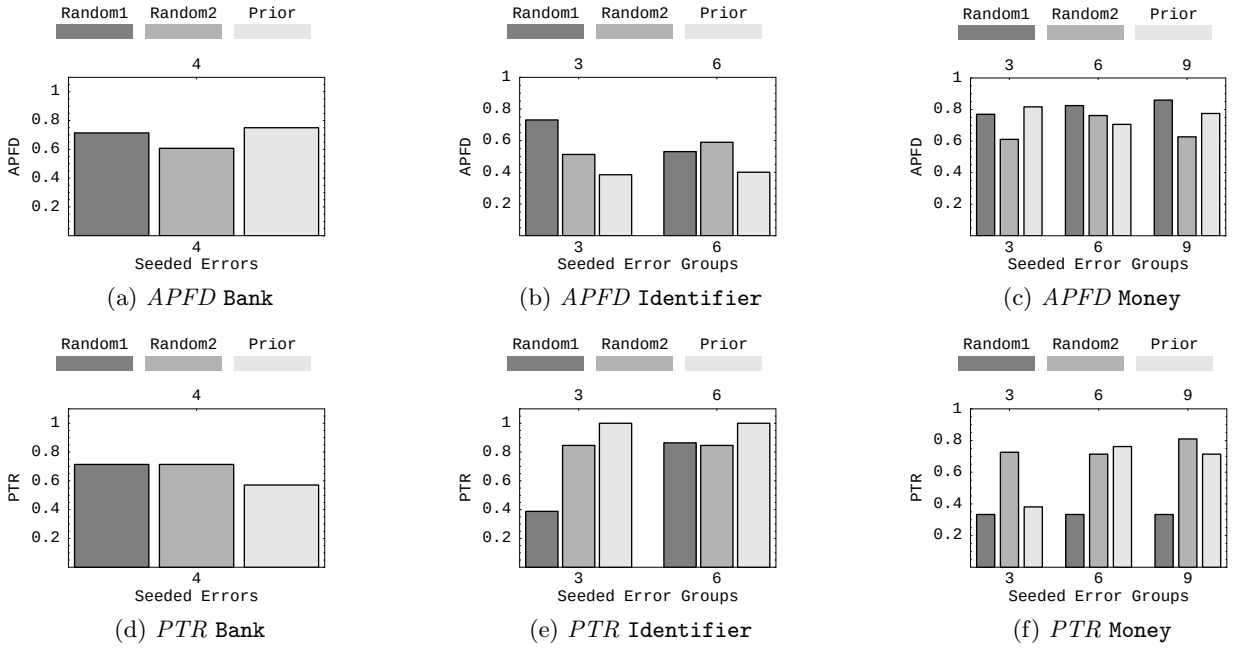
Figure 6 provides the results from our preliminary experiments to measure the effectiveness of *Kanonizo*'s prioritization scheme.[3] In each graph in this figure, the three bars correspond to the values for *APFD* and *PTR* for two random orderings with different seeds (e.g., label "Random1" or "Random2") and the prioritization with respect to *all-DUs* (e.g., label "Prior"). Figure 6(a) and Figure 6(d) show that our technique prioritizes more effectively than random prioritization for the `Bank` application. However, our experiments with the `Identifier` and `Money` applications reveal that *all-DUs* is not always an effective criterion to aid prioritization. For example, Figure 6(b) and Figure 6(e) reveal that our prioritized test suites for `Identifier` perform worse than random prioritization and require the execution of the entire test suite. This is due to the fact that the three test cases with the highest *all-DUs* test adequacy do not reveal any of the seeded defects.

Finally, Figure 6(c) and Figure 6(f) demonstrate that *Kanonizo*'s prioritization can perform better than some random orderings and slightly worse than other random orderings. In light of the knowledge that highly adequate test suites do not necessarily have a high likelihood of revealing a defect [11], these results are not particularly surprising. Yet, since the usage of *Kanonizo* incurs very minimal time and space overheads it can be applied on a per-application basis in order to determine if it is useful. For example, if all or a portion of the defect history of a program is known, our test prioritization approach can be efficiently compared to other techniques in order to determine if it is appropriate to execute the highly *all-DUs* adequate test cases before other tests within the test suite.

8. CONCLUSIONS AND FUTURE WORK

This paper investigates whether the *all-DUs* test adequacy criterion can be used to prioritize the execution of a regression test suite. To this end, we formally describe the *InstrumentAndEnumerate* algorithm that introduces test coverage monitoring instrumentation into the program under test and identifies the test requirements that are necessitated by *all-DUs*. We also provide an equation to calculate the ade-

[3]The experimental results described in this paper focus on a limited number of random prioritizations that typify the results from prioritization experiments. Using the framework described in this paper, we are currently analyzing all permutations of the test suites in order to make statistically significant claims about the effectiveness of prioritization using the *all-DUs* criterion.



(a) *APFD* `Bank` (b) *APFD* `Identifier` (c) *APFD* `Money`

(d) *PTR* `Bank` (e) *PTR* `Identifier` (f) *PTR* `Money`

Figure 6: Measurements of Prioritization Effectiveness.

quacy of a test case and describe the architecture of our prioritization tool called *Kanonizo*. Our experiments reveal that test suites can be prioritized according to *all-DUs* with minimal time and space overhead. However, these preliminary results also indicate that data flow-based prioritizations are not always more effective than random prioritizations. Given the existence of a proven tool that prioritizes according to *all-DUs*, we plan to investigate the following areas: (i) the incorporation of control flow-based and mutation-based adequacy into *Kanonizo*, (ii) the comparison of our prioritization approach to other schemes beyond random, (iii) the calculation of $APFD$ and PTR for all permutations of an application's test suite, (iv) the experimentation with additional case study applications that have larger program segments and test suites, and (v) the investigation of prioritization techniques for test suites that must be executed within a specified time constraint.

9. REFERENCES

[1] Boris Beizer. *Software Testing Techniques*. Van Nostrond Reinhold, New York, NY, 1990.

[2] James Bieman, Sudipto Ghosh, and Roger Alexander. A technique for mutation of Java objects. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, November 2001.

[3] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the 18th International Conference on Software Engineering*, pages 575–584, 1996.

[4] Sebastian Elbaum, Alexey G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, August 2000.

[5] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. Technical Report 03-01-01, Department of Computer Science and Engineering, University of Nebraska – Lincoln, January 2003.

[6] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *International Conference on Software Engineering*, pages 329–338, 2001.

[7] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *J. Syst. Softw.*, 38(3):235–253, 1997.

[8] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[9] Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison Wesley, Boston, MA, 2001.

[10] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 24th International Conference on Software Engineering*, pages 60–71, 2003.

[11] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[12] Gregory M. Kapfhammer. *The Computer Science Handbook*, chapter Software Testing. CRC Press, June 2004.

[13] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2003.

[14] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proceedings of the Twelfth International Symposium on Software Reliability Engineering*, November 2002.

[15] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, pages 201–210. IEEE Computer Society Press, May 1994.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, August 1999.

[17] G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[18] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.